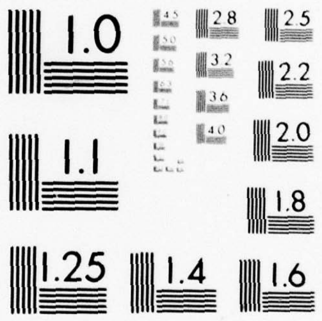
AD-A072 420 NORTHWESTERN UNIV EVANSTON IL DEPT OF ELECTRICAL ENG--ETC F/G 9/2
ON SOFTWARE RELIABILITY MODELING.(U)
JUN 79 S S YAU, T E MACGREGOR F30602-76-C-0397

UNCLASSIFIED RADC-TR-79-127 NL

| OF |
AD
A072420

END
DATE
FILMED
9-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

RADC-TR-79-127
Interim Report
June 1979

ON SOFTWARE RELIABILITY MODELING

Northwestern University

Stephen S. Yau
Terry E. MacGregor

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

APPROVED: Rocco F. Iuorno

ROCCO F. IUORNO
Project Engineer

APPROVED: [illegible]

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

(19) **REPORT DOCUMENTATION PAGE**	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (18) RADC-TR-79-127 — 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER (9)
4. TITLE (and Subtitle) (6) ON SOFTWARE RELIABILITY MODELING.	5. TYPE OF REPORT & PERIOD COVERED Interim Report, 1 Aug 76 — 30 Sep 78,
	6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) (10) Stephen S. Yau Terry E. MacGregor	8. CONTRACT OR GRANT NUMBER(s) (15) F30602-76-C-0397
9. PERFORMING ORGANIZATION NAME AND ADDRESS Northwestern University Department of Electrical Engineering & Computer Science Evanston IL 60201 — MacGregor	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55810278 (16) 5581
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB, NY 13441	12. REPORT DATE (11) June 1979 (17) Ø2
	13. NUMBER OF PAGES 58
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same (12) 55p.	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
Same

18. SUPPLEMENTARY NOTES
RADC Project Engineer: Rocco F. Iuorno (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Software reliability
modeling
figures of merit
time dependent models
time independent models
validation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
In this report, the concept of software reliability is discussed. Various software reliability models along with the basic philosophy of their development, their characteristics, and derived figures of merit are examined. These models' strengths and weaknesses are evaluated. Attempts for validating these models are discussed.

400351 JUB

79 08 06 041

CONTENTS

		Page
	Abstract	ii
	List of Figures	iii
1.	INTRODUCTION	1
1.1	Software Reliability	1
1.2	Survey Objective	2
2.	BACKGROUND	3
2.1	Modeling Process	3
2.2	Model Classification	3
3.	TIME DEPENDENT MODELS	5
3.1	Shooman's Model	7
3.2	De-Eutrophication Model	10
3.3	Geometric De-Eutriphication Model	13
3.4	Geometric-Poisson Model	14
3.5	Shick-Wolverton Model	16
3.6	Weibull Model	17
3.7	Execution Time Model	19
3.8	Bayesian Model	23
3.9	Markov Model	28
3.10	Shooman's Improved Model	30
4.	TIME INDEPENDENT MODELS	34
4.1	Mill's Model	34
4.2	Lipow's Model	35
4.3	Rudner's Model	37
4.4	Nelson's Model	38
4.5	Brown-Lipow Model	40
5.	SUMMARY	43
5.1	Time Dependent Model	44
5.2	Time Independent Model	45
6.	FUTURE RESEARCH	45
7.	REFERENCES	47

LIST OF FIGURES

			Page
Figure	1.	Time Dependent Software Reliability Process	5
	2.	The Correction Process for a Perfect Repair Person	8
	3.	Failure Detection Process	11
	4.	Incidence of Failure Process	11
	5.	Geometric De-Eutrophication Failure Rate	13
	6.	Geometric-Poisson Failure Rate	15
	7.	Schick-Wolverton Failure Rate	16
	8.	Execution Model Correction Process	20
	9.	Failure History of a Program	23
	10.	System Process	25
	11.	Markov Modeling Process	29
	12.	Debugging Process (Imperfect Repair Person - II)	31
	13.	Debugging Process (Imperfect Repair Person - III)	31
	14.	Normal Error Debugging	33
	15.	Unstable Error Debugging	33
	16.	Quasi-Stable Error Debugging	34
	17.	'S' Partition Over Input Space	40
	18.	'Z' Partition Over Input Space	42

1. INTRODUCTION

In the past, the area of software quality assurance has suffered from disorder due to lack of acceptable definitions on what software quality is. Because of the complexity of software, precise definitions of software quality are slow to evolve. Hence, many figures of merit are used to estimate a program's quality. Some of these include efficiency, structure, documentation, reliability, and cost. Figures of merit find their importance during large software development. They supplement the basis from which project managers guide software development.

For large-scale software, cost and reliability are two major areas of concern. In recent years, software systems have become very sophisticated in order to achieve automation of complex physical processes. The cost of development seems to rise exponentially with the size of the program. To better understand this effect, research has been conducted in the area of software reliability in order to develop meaningful figures of merit.

1.1 Software Reliability

Software reliability is an expression that everyone may understand but few can define. Research in this area during the past several years has developed different ways to design reliable software and to estimate its reliability. Design techniques are employed during program specification and coding stages. A collection of techniques for reliable coding, called structured programming, have been formulated. The main idea behind structured programming is to improve reliability by improving understandability and maintainability. Other techniques, developed to aid in program specification, include top-down design and step-wise refinement. Another type of design aid which provides ultra-reliability is the so-called self-checking software. It consists of detecting and containing error propagation by introduction of software redundancies. Design techniques when faithfully followed lead to more reliable software.

Estimation techniques deal with ways to assess a program's reliability when it is able to cycle. Techniques of assessment center around a general definition of reliability proposed by MacWilliams [34]. Reliability is defined as: "The probability that an item will perform its intended function for a specified time interval under stated conditions." This definition emerged from a study which examined the possibility of using the well defined traits of hardware reliability theory to capture directly usable techniques for application to software reliability. Although not entirely successful, the study documented many hardware/software similarities and differences, increased insight into the nature of software reliability, and provided possible directions to pursue.

Figures of merit which have been developed using the above general definition follow two points of view. One viewpoint treats a software system with respect to how it operates upon its input space, which is comprised of all possible data used by a program. The other treats the software system as a block box. From the input space viewpoint, reliability is a measure of the quality of service which is received. More specifically, it deals with a program's capability to produce a correct output for a specific element of the program's input space. Several techniques have been developed to partition a program's input space. Partitions can be user oriented, magnitude oriented, control path oriented, etc. The probability distributions associated with a particular input space are rarely uniform. Hence, when calculating overall program reliability, results must be weighted according to the distribution that governs the input space. This kind of reliability has some additional disadvantages. The failure mechanism which causes an incorrect output for some specific input has several sources. A failure could be induced by a software bug, by hardware failure, compiler failure, etc. Being able to distinguish these failure classes is a nontrivial problem. The main figure of merit derived from this technique is reliability as expressed as the probability that a run of the program will give the correct output for a valid set of input data.

The second viewpoint treats a software system as a black box. The software contains inherent errors which will eventually surface and prevent the program from performing its intended function. Inherent errors are caused by mistakes made during the design and implementation of the program. Reliability from the black box point of view is an inherent property subject to assessment. Several techniques which are primarily modeled after hardware reliability theory have been developed to perform this assessment. The primary weakness of these techniques is the manner by which software errors surface. Hardware reliability theory requires that the failure mechanism be random. In early software debugging, the failure mechanism tend to be systematic. In later debugging stages (especially for large programs), the failure mechanism tends to act in a random fashion. Hence, models which employ the Black Box viewpoint tend to have better accuracy for large programs where the majority of systematic errors have been eliminated. Figures of merit which have resulted include the number of errors inherent to a program, the failure rate, and the mean time to the next failure.

Many models which employ one or the other viewpoints have been developed to estimate reliability and associated figures of merit. Some of these models are fundamental; others are sophisticated. Models have varying degrees of accuracy. This report examines these various models and their ability to estimate accurate figures of merit.

1.2 Survey Objective

The objective of this survey is to provide a tutorial for people who want to become familiar with concepts of this area. Other surveys [9,19, 21,32] in this area have concentrated on validation for comparison purpose or provided very detailed discussion that are only meaningful to an experienced modeler. This report strives for a balance between too much detail and too little explanation. Many models are examined in this report. Previous to actual examination, a background on modeling is given to aid the reader in his understanding. The basic philosophy for model development is examined. Characteristics are defined and used throughout the report. Following this, the models' strengths and weaknesses are evaluated. Finally, attempts for validating these models are discussed.

The models discussed in this report represent the state of the art. In the authors' opinion, the usefulness of reliability models have not been fully exploited. Currently, model validation results imply that many models are data sensitive. Their accuracy is primarily dependent on the skill of the modeler performing the validation.

It is hoped that this report presents a homogeneous picture in enough detail that the reader will appreciate the physical process being modeled and recognize modeling limitations.

2. BACKGROUND

Modeling is a technique of emulating the behavior of a physical system. Many types of models have been used in system studies and have been classified in a number of ways. Most models are classified into physical models or mathematical models. Physical models are such that their attributes are physically measured. A scale model airplane in a wind tunnel is an example of a physical model. Mathematical models, on the other hand, are such that their attributes are represented by mathematical variables. Examples of mathematical models are a characteristic equation of a simple RLC electrical circuit, a statistical model for a sequence of Bernoulli trials, or a Markov model that represents the up-down condition for a two element nonrepairable system.

Software reliability modeling is mathematical. In general, a model is an attempt to utilize mathematics to simulate software that is in steady-state condition. This steady-state condition is one in which a software system is capable of successful operation for extended periods of time. During operation, the software system is susceptible to failure. Failures occur and are corrected. The system executed for a period of time until the next failure occurs.

2.1 Modeling Process

Modeling is basically a three step process. In the first step, the modeler attempts to mold a mathematical function or a set of functions to fit known data points. Once this is accomplished, the second step is to derive some kind of estimator to predict future data points. Data points are generally obtained by observation and seeding. Events are observed by a collector, and seeding is accomplished by stimulating the occurrence of events by inducement. The final step is to validate the model. This process involves collecting data and comparing it to data produced by the model. If a large amount of errors exists in this comparison, then the model is reviewed for validity. Enhancements are made until high correlation exists between actual physical data and modeled data.

Currently, in the area of software reliability modeling, many models have been proposed, but validation is generally lacking, some models have not been validated at all and others have received partial validation.

2.2 Model Classification

Software reliability models which have evolved over the past five years can be classified into those which rely on time as a variable in the modeling process, called <u>time dependent models</u>, and those that do not, called <u>time independent models</u>. Time dependent models look at software systems from the black-box viewpoint. They circumvent consideration of the software's input space. This is done by assuming that error causing elements of a program's input space are uniformly exercised. These models use observation techniques as a means to validate and enhance predictive capability. Software models which fall into this category adapt classical reliability theory as a building foundation. Their accuracy is subject to the randomness by which software errors are discovered.

Time independent models do not emphasize the time aspect of the software system and deal more with the analysis of the program's input space as a means to make reliability predictions. Classical statistical theory is used as a building foundation. Seeding techniques are sometimes used as a stimulus to obtain data to validate and enhance predictive capability. Accuracy of these models is based primarily on the ability to properly partition the input space and to uniformly test all partitions. If the input space is large, sampling criteria must be used to select a representative group of elements from partition. The accuracy of this type of model largely depends upon the sampling criteria used. The models which will be examined in this report are listed in Table 1. They vary in complexity. Some are very basic while others are very sophisticated. The common theme which exists for all these models is that they follow the three step process of generating mathematical functions to known data, deriving an estimator to predict future data, and validating predicted data with observed data.

Table 1 Software Models to be Examined

Time Dependent Models

- Shooman's Model
- De-Eutrophication Model
- Geometric De-Eutrophication Model
- Geometric-Poisson Model
- Schick-Wolverton Model
- Weibull Model
- Execution Time Model
- Baseyian Model
- Shooman's Improved Model

Time Independent Models

- Mill's Model
- Lipow's Model
- Rudner's Model
- Nelson's Model
- Brown-Lipow Model

3. TIME DEPENDENT SOFTWARE MODELS

Time dependent software models can be characterized by those which place emphasis on the detection process [1-11] and those which place emphasis on the detection-correction process [12-17]. In models which place emphasis on the detection process, program failures manifest themselves in a manner analogous to hardware failures with two main distinctions: 1) Software failure is not due to a wearout mechanism, and 2) Once a failure is discovered, it is usually fixed. The correction process which fixes software failures is not considered as an important aspect of the model. Many of the time dependent models of Table 1 fall into this category.

Models which place emphasis on the detection-correction process attempt to more realistically represent the physical picture. In these models, a failure occurs and a correction takes place. In one model [15], the correction process assumes that no new errors are introduced when fixing a failure. In another model [12], an attempt is made to represent the true correction process by accounting for the fact that new errors can be inserted into the software during the repair process.

In general, time dependent software reliability can be represented by the process shown in Figure 1. The number of failures contained in a program is

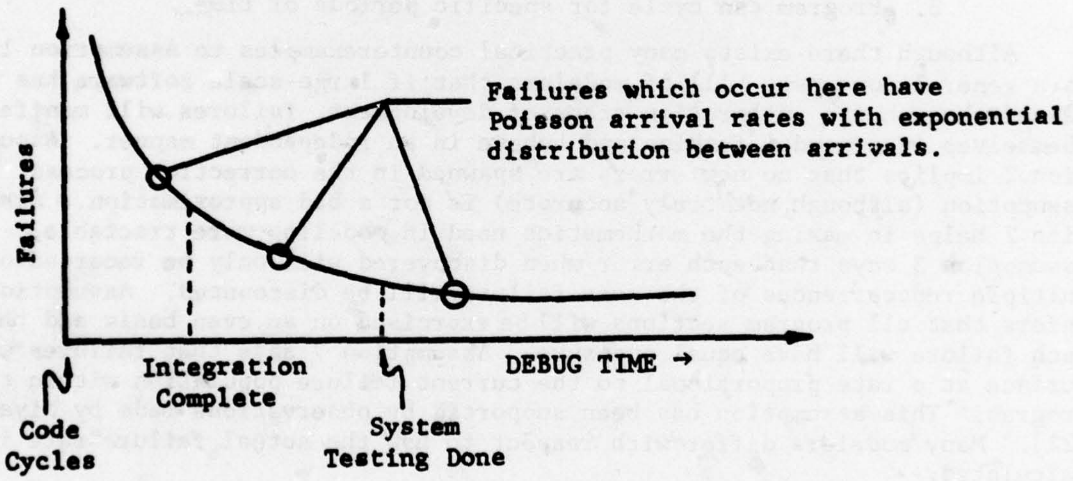

Failures which occur here have Poisson arrival rates with exponential distribution between arrivals.
Failures
DEBUG TIME →
Code Cycles
Integration Complete
System Testing Done

Figure 1 Time Dependent Software Reliability Process

proportional to the number of lines of code and tend to diminish to a specific level as a function of debug time. Debug time is defined to exist whenever a software system is running and failures are being detected. This time frame essentially covers the life cycle of the program. Debug time can be measured as CPU execution time or calendar time. Failures which occur during debug time are assumed to have Poisson arrival rates with an exponential distribution between arrivals.

Time dependent models which emphasize the detection process (explicitly or implicitly) make the following basic assumptions:

1. Failures are independent.
2. The number of failures in the program is constant.
3. Each failure is repaired before testing continues.
4. Inputs which exercise the program (whether testing or operational) are randomly selected.
5. All failures are observable.
6. Testing is of uniform intensity and representative of the operational environment.
7. The failure rate at anytime is proportional to the current failure constant.
8. Program can cycle for specific periods of time.

Although there exists many practical counterexamples to Assumption 1, it is a general consensus [11] of modelers that if large-scale software has passed through its integration stage of development, failures will manifest themselves in a random fashion and behave in an independent manner. Assumption 2 implies that no new errors are spawned in the correction process. This assumption (although not truly accurate) is not a bad approximation. Assumption 2 helps in making the mathematics used in modeling more tractable. Assumption 3 says that each error when discovered will only be recorded once. Multiple reoccurrences of the same failure will be discounted. Assumption 6 infers that all program sections will be exercised on an even basis and that each failure will have equal exposure. Assumption 7 says that failures will surface at a rate proportional to the current failure population within the program. This assumption has been supported by observations made by Miyamoto [22]. Many modelers differ with respect to how the actual failure rate is calculated.

The probability of failure of a software system which satisfies these basic assumptions can be calculated using standard textbook theory [18]. The probability of software failure can be expressed as

$$P\{t < t_f \leq t + \Delta t \mid t_f > t\} = Z(t)\Delta t, \tag{1}$$

where $P\{t < t_f \leq t + \Delta t \mid t_f > t\}$ is the probability that a failure will surface and be discovered in time interval t to t + Δt given that no failure have occurred until time t, and Z(t) is to the failure rate. Assuming that the interarrival times of failures are exponentially distributed, the software reliability R(t) which is the probability that no errors have occurred within a given time period, is given by

$$R(t) = \mathrm{EXP}(-\int_0^t Z(t)dt) \quad (2)$$

The mean time to failure (MTTF) is given by

$$\mathrm{MTTF} = \int_0^\infty R(t)dt \quad (3)$$

Equations (1) to (3) are applicable to all the time dependent software models in Table 1 with the exceptions of the Markov and Bayesian models.

The primary distinction between models which are represented by the general equations (1) to (3) are the various failure rates Z(t) used. Each software model approximates the failure rate in a different manner. We are going to examine each of these models in more detail in the following sections.

3.1 Shooman Model

The Shooman Model [1-5] is one of the earliest models developed to predict software reliability. It was conceived in 1971 and published again in 1972, 1973, and 1975. The major assumption made by the Shooman model is that the failure rate is proportional to the number of failures residing in the software at any time. Z(t) can be expressed

$$Z(t) = C\varepsilon_r(\tau) , \quad (4)$$

where C is a constant of proportionality, t is program execution (CPU) time, τ is debugging time (in months), and $\varepsilon_r(\tau)$ is the error rate which is the number of errors remaining in the program at time τ and normalized with respect to the total number to instructions. $\tau \gg t$ hence events which vary with debugging time appear to be constant with respect to execution time. $\varepsilon_r(\tau)$ can be expressed as

$$\varepsilon_r(\tau) = \frac{E_T}{I_T} - \varepsilon_c(\tau), \quad (5)$$

where

I_T = number of program instructions (constant)

E_T = number of failures present when debugging begins

$\varepsilon_c(\tau)$ = number of errors fixed in interval 0 to τ (normalized).

Knowing the form of the failure rate, reliability and MTTF become

$$R(t) = \mathrm{EXP}[\ -C(\frac{E_T}{I_T} - \varepsilon_c(\tau)t)] \quad (6)$$

$$MTTF = \frac{1}{C\left[\frac{E_T}{I_T} - \varepsilon_c(\tau)\right]} \qquad (7)$$

The accuracy of this model depends on several factors. Two factors involve the basic assumptions that the number of errors are fixed and that the failure rate is proportional to the remaining failures. The fixed error assumption is the weaker of the two assumptions, and implies the existence of a perfect repair person who does not spawn new failures while performing the correction procedure. Debugging using the perfect repair person yields a correction process which is depicted in Figure 2

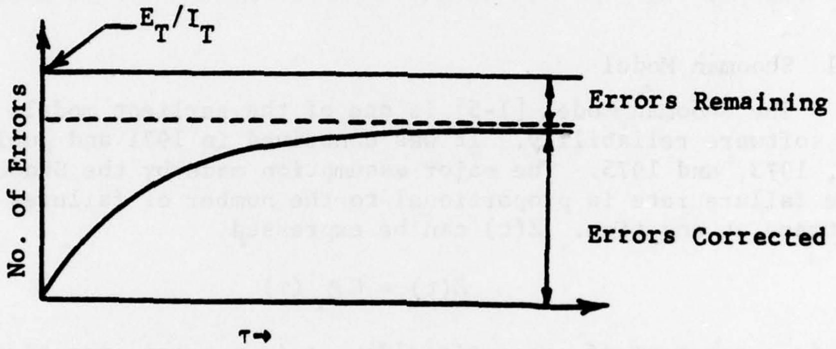


Figure 2. The Correction Process for a Perfect Repair Person

Another assumption which affects the accuracy of this model is the repair and correction rate. It was assumed that the repair and correction rate was a function of calendar time. This assumption is somewhat disputed by similar models [10,11] of later vintage. These models assume that the repair and correction rate is a function of CPU execution time.

Accuracy is also affected by estimators required by the model. Two techniques are developed [4] to estimate the total number of failures E_T initially present and the constant of proportionality C. The moment technique requires that a functional test must be run on the software under test at two different times. The times should be chosen so that an adequate amount of debugging has occurred. Data collected at these times include the number of software failures and the amount of successful run hours. The MTTF is calculated for these two known points using:

$$MTTF = \frac{1}{\lambda} = 1/\frac{\#\text{ of failures}}{\text{successful run hours}} \qquad (8)$$

Using (7) and (8), the estimated $\hat{E}_T$ and $\hat{C}$ are given by

$$\hat{E}_T = \frac{I_T\left[(\lambda_{s_2}/\lambda_{s_1})\varepsilon_c(\tau_1) - \varepsilon_c(\tau_2)\right]}{(\lambda_{s_2}/\lambda_{s_1}) - 1} \tag{9}$$

$$\hat{C} = \frac{\lambda_{s_1}}{\frac{\hat{E}_T}{I_T} - \varepsilon_c(\tau_1)}\ , \tag{10}$$

where $\lambda_{s_i} = \dfrac{\underline{\#\text{ of software failures during test time } i}}{\text{successful run hours during test time } i}$ $i = 1,2$

The *second* technique, <u>Maximum Likelihood Estimator</u> used to estimate C is believed by many to be more accurate than the moment approach. This technique is discussed in detail in reference 17 and applied to a similar model in reference 6. The estimated E_T is found to be the same as that obtained by the moment technique, that is given by (9). The estimated C becomes

$$\hat{C} = \left[\frac{1}{H_1 + H_2}\right]\left[\frac{n_1}{\frac{\hat{E}_T}{I_T} - \varepsilon_c(\tau_1)} + \frac{n_2}{\frac{\hat{E}_T}{I_T} - \varepsilon_c(\tau_2)}\right] \tag{11}$$

where H_i = successful run hours $i = 1,2$

n_i = number of runs $i = 1,2$

Inaccuracy can also be caused by the form which $\varepsilon_r(\tau)$ assumes. (4) assumes that the failure rate is proportional to the remaining errors. In the model discussed so far, $t \ll \tau$ hence R(t) is calculated as if $\varepsilon_c(\tau)$ were constant. A more accurate expression would be

$$Z(t) = C\left[\frac{E_T}{I_T} - \int_0^{\tau} p(x)dx\right] , \qquad (12)$$

where p(x) represents the error detection rate per instruction as a function of calendar time. A study [3] was made to investigate the shapes that p(x) might exhibit. The study concluded that p(x) took on a triangular form for one set of data and a rectangular form for another set of data.

Several objective attempts [10,19,21] to validate this model have been made. Wagoner's attempt [10] yielded inconclusive results. Calculating the proportionality constant C using the moment and maximum likelihood estimator technique yielded significantly different estimations. Sukert's [19, 20] collected data was not too useful in this case because they did not support the CPU time requirement. Stucki [21] was able to use the model to calculate MTTF for data defined in reference 10. The data was applied against the model in slightly adjusted form. Zero time was shifted to a point more suitable for all models under test. The main rationale was to eliminate transit data caused by start-up effects. Stucki observed that estimates of E_T using (6) vary as the time τ_1 and τ_2 between tests increase.

3.2 De-Eutrophication Model

The De-Eutrophication Model [6,7] was developed in the same time frame as the Shooman model. It was developed to estimate the initial (or residual) failure content in a large software program. In addition, it could estimate time between failure during any part of the test phase.

The major assumption, in addition to the basic assumptions discussed before, is that the failure rate be modeled by

$$z(t_i) = \phi\ [N - (i-1)] \qquad (13)$$

This process is depicted in Figures 3 and 4. In Figure 3, at time t_0, the

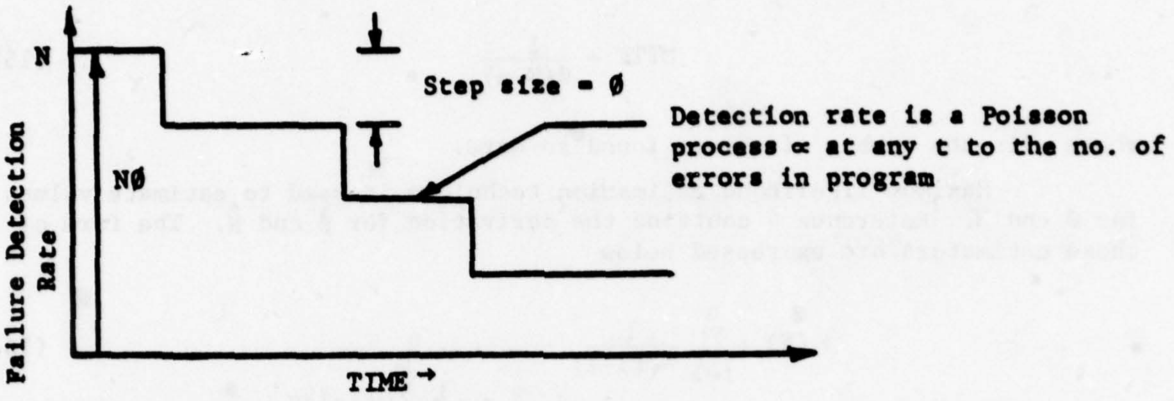

N
NØ
Step size = Ø
Detection rate is a Poisson process ∝ at any t to the no. of errors in program
Failure Detection Rate
TIME →

Figure 3 Failure Detection Process

program contains N estimated failures. Each failure-detection decreases the failure rate in a linear stepwise (NØ) fashion, where Ø is the stepwise constant. During the interval between failures, the failure rate is assumed to be a Poisson process proportional at any time to the number of failures in the program. Figure 4 shows the corresponding time periods associated with each failure. A value associated with t_i is the time (either calendar or CPU) in which the program successfully runs between failures i-1 and i. These times are assumed to be statistically independent.

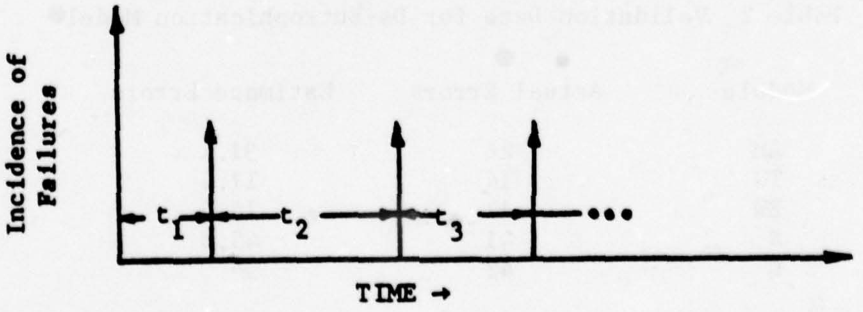


Figure 4 Incidence of Failure Process

Reliability and MTTF for this model are

$$R(t_i) = \text{EXP}\left[-\phi(N-n)t_i\right] \tag{14}$$

$$\text{MTTF} = \frac{1}{\phi(N-n)} \tag{15}$$

where n is the number of errors found to date.

Maximum likelihood estimation technique is used to estimate values for ϕ and N. Reference 6 contains the derivation for $\hat{\phi}$ and $\hat{N}$. The form of these estimators are expressed below

$$F(N) = \sum_{i=1}^{n} \frac{1}{N-(i-1)} - \frac{n}{N - \frac{1}{T}\sum_{i=1}^{n}(i-1)t_i} \tag{16}$$

$$T = \sum_{i=1}^{n} t_i \tag{17}$$

$$\hat{\phi} = \frac{n}{\hat{N}T - \sum_{i=1}^{n}(i-1)t_i} \tag{18}$$

(16) must be solved numerically for a meaningful value of N which causes F(N)=0. Once determined, N can be used to calculate ϕ.

This model was validated by Jelinski and Moranda [6] on five software modules. Table 2 summarizes their results.

Table 2 Validation Data for De-Eutrophication Model

Module	Actual Errors	Estimate Errors
AM	26	31.1
TW	14	17.4
EW	12	14.4
S	41	45.8
C	42	54

Another validation study [19] revealed that this model has consistently higher predictions for the number of remaining errors than the actual data showed. The predicted values were more accurate when the time interval between failures was measured as weeks. The starting time t_0 also appears to affect accuracy. Estimation of N and ϕ failed to work properly for some of the data tested.

Stucki [21] also observed the starting time t_0 phenomenon and circumvented the problem by eliminating the first several observations of failing times, and hence shifting the starting position of t_0. Failures which had accumulated up to that point were added in with the initial estimates of N. This technique produced reasonable estimations of MTTF.

Miyamoto [22] has also published data confirming the de-eutrophication process, along with the maximum likelihood estimation of its parameters.

A strength of this model appears to be its ability to make reasonable estimations for various classes of data. Time can be measured with respect to calendar time or CPU execution time. A weakness of this model is its sensitivity to data as observed by both Sukert [19] and Stucki [21].

3.3 Geometric De-Eutrophication

The Geometric De-Eutrophication Model [8] employs the same basic process as the De-Eutrophication Model, except that the step size of its step-wise failure rate is a geometric progression. The interval at any time between failure is still assumed to be a Poisson process proportional at any time to the number of failures in the program. Interval times are governed by the detection of failures. The failure rate is depicted by the process shown in Figure 5, in which D is the initial detection rate and k is the geo-

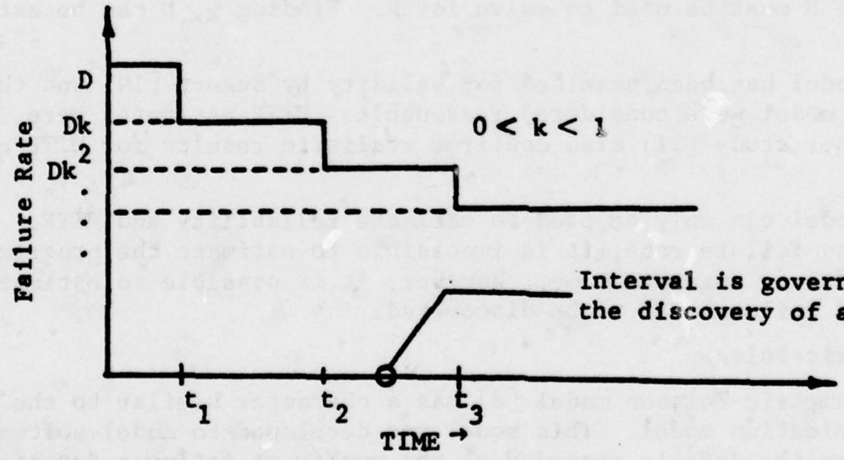


Figure 5 Geometric De-Eutrophication Failure Rate

metric constant. The value of k is within the range $0 < k < 1$ so that rates form a converging geometric series. t_i is the time interval between the (i-1)th and i the failures. The failure rate is given by

$$Z(t_i) = Dk^{i-1} \tag{19}$$

Reliability and MTTF are

$$R(t_i) = EXP[-Dk^n t_i], \tag{20}$$

where n is the number of failures discovered to date

$$MTTF = 1/Dk^n \tag{21}$$

Maximum likelihood estimators are used to predict k and D. Derivations utilizing the maximum likelihood estimation technique can be found in reference 8. The estimation expressions are given by

$$F(k) = \frac{\sum_{i=1}^{n} ik^i t_i}{\sum_{i=1}^{n} k^i t_i} - \frac{n+1}{2} \tag{22A}$$

$$\hat{D} = \frac{n}{\sum_{i=1}^{n} \hat{k}^{i-1} t_i} \tag{22B}$$

A numerical approach similar to the approach used in the De-Eutrophication model to estimate N must be used to solve for $\hat{k}$. Finding $\hat{k}$, $\hat{D}$ can be estimated.

This model has been examined for validity by Sukert [19] and the results for this model were considered reasonable. MTTF estimates were realistic. Another study [21] also confirms realistic results for MTTF estimation.

This model can only be used to estimate reliability and MTTF. Due to the form of its failure rate, it is impossible to estimate the program's failure content at any point in time. However, it is possible to estimate the percentage of failures yet to be discovered.

3.4 Geometric-Poisson

The geometric-Poisson model [8] has a character similar to the geometric de-eutrophication model. This model was developed to model software development, where the data is recorded as the number of failures detected per/month. Figure 6 depicts this process.

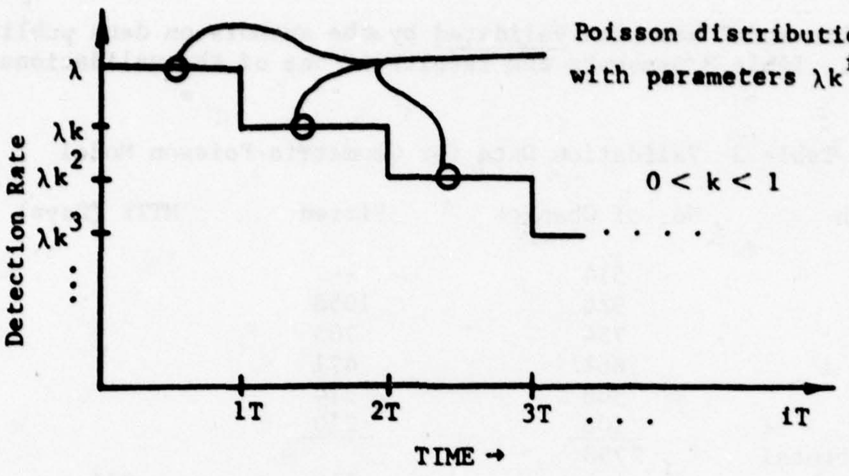


Figure 6' Geometric-Poisson Failure Rate

The similarity between this and the geometric de-eutrophication process is that the detection rate decreases in a geometric manner. Instead of decreasing each time a failure occurs, the geometric-Poisson failure rate decreases at fixed intervals. λ is the average number of failure detected during the first period, and k is the geometric proportionality constant. The failure rate is expressed as

$$Z(t_i) = \lambda k^{i-1} \tag{23}$$

which says the average number of failures detected in post intervals is proportional to the number detected in the first interval. Reliability and MTTF can be expressed as

$$R(t_i) = \mathrm{EXP}[-\lambda k^{i-1}] \tag{24}$$

$$\mathrm{MTTF} = \int_0^{\infty} R(t_i)dt_i \tag{25}$$

λ and k can be predicted using the maximum likelihood estimation technique similar to that used on other de-eutrophication models. They can be obtained by solving the following equations

$$\frac{1}{\lambda} \sum_{i=1}^{m} n_i = \sum_{i=0}^{m-1} k^i \tag{26}$$

$$\lambda \sum_{i=0}^{m-1} ik^i = \sum_{i=0}^{m-1} in_{i+1} \tag{27}$$

where m is the number of observations and n_i is the number of errors observed during fixed time interval i.

This model has been validated by the authors on data published in reference 8. Table 3 presents the results of one of the validations.

Table 3 Validation Data for Geometric-Poisson Model

Month	No. of Changes	Fitted	MTTF (Days)
1	514	--	
2	926	1058	
3	754	705	
4	662	471	
5	308	314	
6	108	210	
Subtotal	2758		
7		140	.214
8		93	.323
9		62	.484
		3174	

Applying the model to the data of Table 3 requires compilation of a program failure profile. The profile is used to estimate initial values of $\hat{\lambda}$ and $\hat{k}$. A curve fitting process is used to refine the values of $\hat{\lambda}$ and $\hat{k}$. Prediction can be made by using the refined estimators.

Another study [21] produced predictions for its data that were less accurate for all the models compared.

3.5 Schick-Wolverton Model

The Schick-Wolverton model [9] follows the same development of the de-eutrophication model. The primary difference is the failure rate, which is depicted by Figure 7. The failure rate is proportional to the number of remaining errors and increases with operating time. Operating time is defined to start at interval t_{i-1} and end at t_i. The failure rate is given by

$$Z(t_i) = \phi[N - (i-1)]t_i, \tag{28}$$

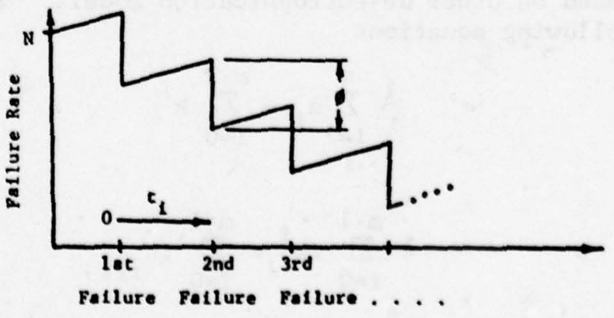


Figure 7 Schick-Wolverton Failure Rate

where t_i are statistically independent and define the time between the (i-1)th and ith failure. The reliability and MTTF are given by

$$R(t_i) = \mathrm{EXP}[-\emptyset(N-n) \frac{t_i^2}{2}] \quad (29)$$

$$\mathrm{MTTF} = [\pi/2\emptyset\ (N-n)]^{\frac{1}{2}} \quad (30)$$

where n is the number of errors found to date. Ø and N are estimated using maximum likelihood estimators. The solution technique is similar to that employed in reference 6. The likelihood equations are

$$\frac{2n}{\emptyset} = \sum_{i=1}^{n} [N-(i-1)]\, t_i^2 \quad (31)$$

$$\sum_{i=1}^{n} \frac{1}{N-(i-1)} = \frac{\emptyset}{2} \sum_{i=1}^{n} t_i^2 \quad (32)$$

Several validations [10,19,21] of this model have been made. Wagoner [10] uses this model to compute the distribution of failures for his data. The model produced results that were consistently less than actual results for very early testing times. For later test times, the model yielded results that were higher. Overall, Wagoner's results were reasonable. Stucki's results were inconclusive [21]. The initial estimation of N was less than those produced by other models. A conclusion was made that the Schick-Wolverton model did not appear to fit the error process very well. Wagoner and Stucki used the same data. An observation made by Sukert [19] showed that the Schick-Wolverton model consistently gives higher predictions than that in actual data. Variations from the expected results were large. It has also been observed that the model is more accurate at predicting the number of residue failures for smaller groups of data than larger groups. The model appears less accurate than the de-eutrophication model when time intervals were measured in days. The model observed the same estimation problems previously discussed for the de-eutrophication model.

A primary weakness of this model is its high sensitivity to data. To obtain reasonable results, data must be adjusted on an individual basis by an experienced modeler.

3.6 Weibull Model

The Weibull model [10] is a time dependent model which assumes a failure rate of the form:

$$Z(t) = \frac{\eta}{\sigma} \left(\frac{t}{\sigma}\right)^{\eta-1} \quad (33)$$

where η is the shaping parameter and σ is the scale parameter. This failure rate has the unique capacity to control the rate at which failures occur. If $\eta=1$, then the failure rate becomes constant, and the Weibull model reduces to Shooman type models [3,6,11]. If $\eta > 1$, then the failure rate increases, and the model becomes unrealistic. Software which has $\eta > 1$ never completes the testing phase, and never becomes operational because failures of the program grow with time. If $\eta < 1$, the failure rate decreases with time, and software can become operational in a less amount of time. The reliability and MTTF based on this failure rate are given by

$$R(t) = \text{EXP}\left[-\left(\frac{t}{\sigma}\right)^{\eta}\right], \tag{34}$$

$$\text{MTTF} = \int_0^{\infty} R(t)dt = \frac{\sigma\,\Gamma\left(\frac{1}{\eta}\right)}{\eta} \tag{35}$$

The Weibull model was developed as an effort to extend software reliability modeling for better accuracy. An experiment was conducted to compare existing models [4,6,9] against known published sources of data [23-25]. The result of the study was rather inconclusive. The Weibull scale and shape parameters were fitted to the above data source. All fits were unsuccessful because they yielded values of $\eta > 1$. As previously discussed, this is impractical. Wagoner [10] speculates that the unsuccessful fits were due to the time reference recorded when failures occurred. Time was measured with respect to calendar days. To prevent the collaspe of the experiment, data was gathered on an internal program. Failures were recorded as functions of CPU time and 108 failures were observed over a 19 day (226.1 CPU seconds) test period. Parameters (e.g. Shooman's constant of proportionality, Weibull's shape and scaling factors etc.) were estimated for each model. Some discrepancies were observed. For example, calculation of parameters for Shooman's model using prescribed criteria yielded estimations of significantly different values. Discrepancies were observed for the de-eutrophication model. The source of these discrepancies was not isolated.

To circumvent these problems, the decision was made to calculate the affected parameters by using the internally gathered data and the general definition of the failure rate

$$Z(t_i) = \frac{n(t_i) - n(t_i + \Delta t_i)}{n(t_i)\Delta t_i} \tag{36}$$

The parameters were calculated using the following formulae:

Shooman $$C = \frac{Z(t_i)}{\varepsilon_r(\tau)} \tag{37}$$

De-Eutrophication $$\phi = \frac{Z(t_i)}{N-(i-1)} \qquad (38)$$

Schick-Wolverton $$\phi = \frac{Z(t_i)}{N-(i-1)t_i} \qquad (39)$$

This technique also yielded inconclusive results because all constants (which are suppose to be stationary) varied with time. Hence, in the final comparison, only the actual cummulative distribution of failure was compared with data produced by the Schick-Wolverton [9] and Weibull modeling processes. The Weibull process more faithfully reproduced the cummulative distribution curve. It is reasonable for this to occur since this model has an extra degree of control.

Inconclusiveness is the conclusion that can be derived when considering whether the Weibull model furthers the state of art of software reliability modeling. For one set of internally generated data, the Weibull model appears to produce more accurate results than did the Schick-Wolverton model.

3.7 Execution Time Model

The Execution Time Model [11] incorporates concepts of previously discussed models to produce a more general model. The development defines a relationship between CPU execution time and calendar time, and shows that the correction rate varies as an exponential curve with execution time while taking on various forms in calendar time. This model makes all the basic assumptions and assumes that failures are distributed at any time with a constant average occurrence rate. It makes a slight modification of the error spawning assumption.

In the <u>general model development</u>, the model assumes that at any point in time the failure rate is proportional to the residue failures in a program and has the following form

$$Z(\tau) = fk\Omega \qquad (40)$$

where f is the linear execution frequency, which is the instruction processing rate divided by the total number of instructions, k relates the error exposure frequency to the linear execution frequency and Ω is the number of failure still residing in the program. Exercising the constant failure population assumption, the failure rate can be expressed in terms of the total failures N_0 and corrected failures n as follows:

$$Z(\tau) = fkN_0 - fkn \qquad (41)$$

By expressing f and n as defined by Shooman, (41) can be shown to be equivalent to the Shooman failure rate.

Using the following basic assumptions

1. All failures are observed,
2. Failures are fixed before testing continues, and
3. No new failures are spawned in the correction process,

the correction rate can be assumed equal to the failure rate, that is

$$\frac{dn}{d\tau} = Z(\tau) \qquad (42)$$

Solving this equation yields:

$$n = N_0 (1 - \mathrm{EXP}(-fk\tau)) \qquad (43)$$

which is depicted by Figure 8 .

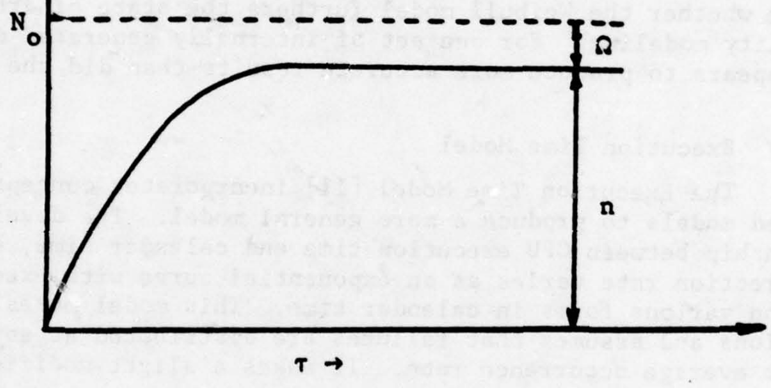


Figure 8 Execution Model Correction Process

The execution time model yields the same basic curve as Shooman's model. MTTF is given by:

$$\mathrm{MTTF} = \frac{1}{fkN_0} \mathrm{EXP}(fk\tau) \qquad (44)$$

It is seen that the basic execution model is equivalent to the Shooman model, as previously discussed.

Generalization can be made to this model by adding an error reduction factor B and compression factor C. B is the ratio of error reduction to error occurrence. C is the ratio of detection rate during test to that during time of operation. The correction rate now becomes

$$\frac{dn}{d\tau} = BCZ(\tau) \qquad (45)$$

Since the execution time model assumes that each error is fixed, B implies that errors are spawned on an average rate. While this may not be a bad assumption, the spawning effect is not fed back to correct the total number of failure still residing in the program. In reality, B accounts for the differences between the total number of failures inherent in the software and the number of failures required to expose and remove these inherent failures. Empirical evidence shows that B ranges from .91 to .99. B can b estimated with medium accuracy.

C accounts for the fact that testing shakes out more failures than operational exercise does. This parameter is usually estimated based on experience. The accuracy of the estimation is low. Empirical values of C range from 1 to 10.

The basic model equations (43) and (44) can be expressed in terms of the number of failures experienced in correcting n errors and the number of failures required to expose and remove N_0 inherent errors. Then,

$$m = M_0(1-\mathrm{EXP}(-BCfk\tau)), \tag{46}$$

where $m = n/B$ and $M_0 = N_0/B$

$$\mathrm{MTTF} = T_0\mathrm{EXP}(BCfk\tau), \tag{47}$$

where $T_0 = 1/BfkM_0$

Maximum likelihood estimation techniques are used to predict values of M_0 and T_0. The derivation of these estimators and their accuracy are found in appendix B of reference 11.

Development of a Calendar Time - Execution Time Relationship was stimulated out of the desire to account for how computer resources effect testing. In past research [1-5], testing was assumed to be uniform and the failure detection rate constant but experience has shown that testing influences the failure detection capability, and the rate at which failures can be discovered. This in turn affects model accuracy by partially invalidating the fundamental assumptions on which the model is based.

The rate of testing is constrained by three resources: computer time (t_C), people who perform failure identification (t_I), and repair personnel (t_F). At any point in calendar time t, one of the three resources will limit the other two resources. For example, consider the case where failures are found and repairs are prepared. If the computer is unavailable, the repairs cannot be tested. Hence the identification and repair are constrained by the computer resource (t_C). In another example, a back log of failures can exist. The computer is available. The failure repair resource now constrains the other resource. Now by letting $\frac{dt_I}{d\tau}$, $\frac{dt_C}{d\tau}$ and $\frac{dt_F}{d\tau}$ be a measure of the amount that a resource constrains the testing process at any instant of execution time, an increment of calendar time Δt can be defined to be proportional to the average amount by which the dominant resource constrains testing over a given execution time segment. More formally Δt can be expressed as

$$\Delta t = \int_{\tau_1}^{\tau_2} \max\left(\frac{dt_I}{d\tau}, \frac{dt_F}{d\tau}, \frac{dt_C}{d\tau}\right) d\tau \qquad (48)$$

This is the fundamental definition of the calendar time - execution time relationship.

The amount by which a resource affects the testing process can be minimized by predicting the resource requirements. According to Musa [11], a model which closely approximates all three resource requirements has the following forms

$$\chi = \theta\Delta\tau + \mu\Delta m \qquad (49)$$

$$\chi = \rho P \Delta t \qquad (50)$$

where χ = resource requirement

θ = average expenditure rate with respect to execution time

μ = average resource expenditure

$\Delta\tau$ = basic unit of CPU time

Δm = number of failures which must be detected to improve MTTF by some quantum amount

$P = \dfrac{\text{computer availability}}{\text{personnel to utilize computer}}$

ρ = utilization factor

A derivation [11] which incorporates the resource model expressed by (49) and (50) into the execution time - calendar time relationship shown in (48) yields

$$\Delta t = \frac{1}{BCfk} \int_{t_1}^{t_2} Z(\tau) \max_k \left[\frac{\theta_k}{P_k \rho_k} + \frac{C\mu_k}{P_k \rho_k}, Z(\tau)\right] d\tau \qquad (51)$$

where C, F, or I may be k for each segment k. This implies that on the average for any given execution time segment, the calendar time increment is proportional to the number of expected failures and the amount by which testing resources constrain the testing process. In other words, if the software has a lot of errors and testing is being constrained by resources, then the time required to pass the software system through its test phase will be long.

This model has been validated by Musa [11] on four medium-size software projects which ran several months in duration. Modeling results for project 1 were included as part of the paper. Verification was made by recording the actual number of failures detected and the execution time of their detection. The cummulative number of failures was transformed using $-\ln(1-m/M_0)$, and plotted against the execution time. M_0 was estimated at the end of testing from data gathered throughout testing. The transformation was derived from (46). The curve $\ln(1 - \frac{m}{M_0}))$ vs τ represents the theoretical limit which the actual data should come close to representing. Correlation was high for Project 1. The Calendar Time-Execution Time relationship was validated in an analogous manner.

A unique measurement was made to evaluate the robustness of the modeling technique. M_0 and T_0 were constantly reestimated as execution time continued and plotted as a function of execution time. The series of estimations were pretty much a constant as execution time increased. This is consistent with the theoretical result. T_0's estimate grew with the execution time that represented early testing and then leveled off later in the testing phase. The growth was attributed to the fact that testing overlapped with integration; and hence, testing was not distributed across all areas of the program.

Other data plotted was derived from the real data and tended to verify physical processes that software testing has known to exist (e.g. MTTF increases with test time, the number of failures required to cause significant increase in MTTF gets less as test time continues, etc.) The data presented was representative of the other three projects. This would tend to infer that the execution time model is one of the most accurate software reliability models currently in existence.

3.8 Bayesian Model

The Bayesian Model [12, 13] is a time dependent model having unique characteristics. It approaches the modeling process in a probabilistic rather than deterministic manner. In doing so, it eliminates the "error spawning" assumption which applies to all models so far discussed. Past models assume the existence of a perfect failure repair person. The Bayesian approach circumvents this assumption, creating a repair rule to approximate a correction process as executed by an imperfect repair person.

The primary assumptions made in this model are

1. The program is large and can be viewed as a black box.
2. The program executes as a continuous time process.
3. All error correction occurs instantaneously.

The modeling process examines the program's failure history as shown in Figure 9.

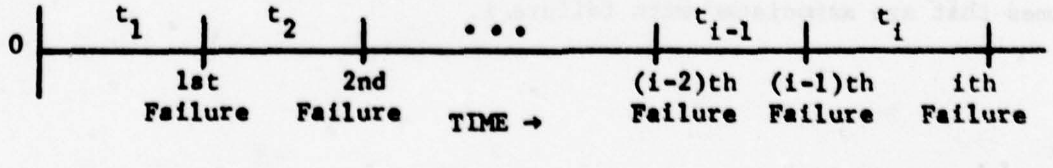


Figure 9 Failure History of a Program

At t_0 the program can at least cycle for some period of time. t_1 is the time that has elapsed between program startup (t_0) and the first failure. t_2 is the time that elapses between the first and second failures, etc. Associated with each time segment is a failure rate λ. If repair of failure were affected by a perfect repair person, then the theoretical repair rate, which is given below should be used:

$$\lambda(i) < \lambda(i-1) \tag{52}$$

Physical interpretation of (52) says that as program testing continues with time, the failure rate should decrease with time. This seems intuitively correct.

The Bayesian model contends that perfect repair personnel do not exist. Therefore, at best, the failure rate remains constant with time, or more likely increases with time due to new failures spawned during the correction process. This contention can be modeled by assigning probabilities to each time interval as follows:

$$P\{\lambda(i) < \ell\} \geq P\{\lambda(i-1) < \ell\}, \tag{53}$$

for every i and ℓ. Interpretation of (53) for the first two time intervals implies that the probability that $\lambda(2)$ will be less than some constant is greater than (the more likely case) and equal to (the best case) the probability that $\lambda(1)$ will be less than the same constant. $P\{\lambda(i) < \ell\}$ can be calculated by

$$P\{\lambda(i) < \ell\} = \int_{-\infty}^{\ell} g(\ell,i,\alpha)d\ell \tag{54}$$

where $g(\ell,i,\alpha)$ is the probability density function which governs the influence that the imperfect repair person has over λ. The form of $g(\ell,i,\alpha)$ is unknown. However, $g(\ell,i,\alpha)$ is known to be a function of some upper limit ℓ which λ can reach, the time interval i, and a parameter α representing the confidence in the correction process.

The system process which depicts Figure 11 and (53) and (54) is shown in Figure 10. Given some confidence factor α, the failure rate $\lambda(i)$ for a specific interval can be defined. The failure rate defines the next program failure t_i which can be observed. Observation of several t_i forms a statistic called <u>data</u>. Using this process the Bayesian model predicts the time interval for the i+1 failure. Prediction of the T_{i+1} time interval centers around the confidence factor α and the distributions which govern the A, F.R., and T sets of Figure 10, where set A is the set of all possible confidence factors that α can be, set F.R. is the set of all possible interfailure times that are associated with failure i.

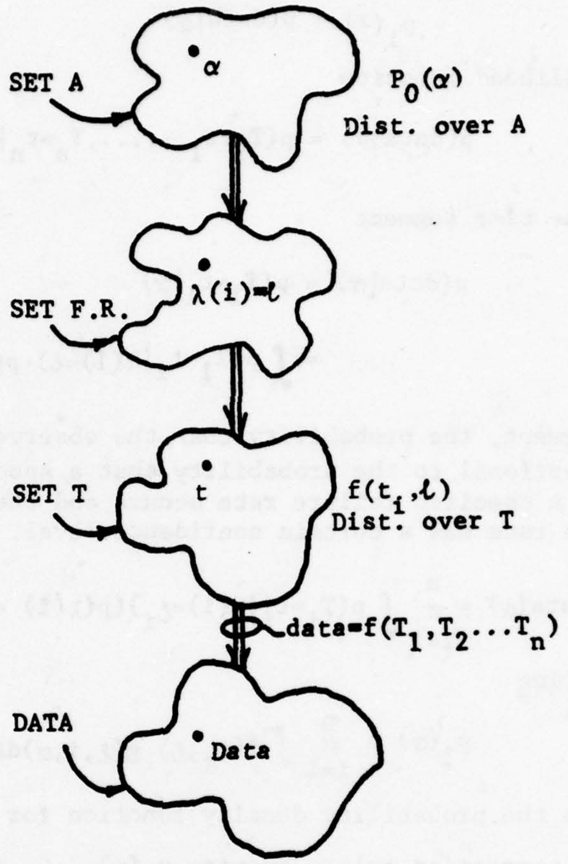


Figure 10 System Process

Assuming that the data has been collected, then the probability that the data observed has a certain confidence factor can be expressed by the Bayesian expression

$$p_1\ (\alpha|\text{data}) \propto p(\text{data}|\alpha)p_o(\alpha) \tag{55}$$

By making an assumption about the distribution over A, $p_1(\alpha|\text{data})$ becomes a more accurate estimation of the actual distribution of A. $p_o(\alpha)$ is called the prior distribution, $p(\text{data}|\alpha)$ is called the likelihood function and $p_1(\alpha|\text{data})$ is called the posterior distribution. Assuming that the posterior distribution is representative of the actual distribution over A and the prior distribution is uniform, then the distribution of A is proportional to the likelihood function. Assuming that $p_o(\alpha)$is a uniform distribution and

$$p_1(\alpha) = p_1(\alpha|\text{data}), \tag{56}$$

then

$$p_1(\alpha) = p(\text{data}|\alpha) \tag{57}$$

To find the likelihood function

$$p(\text{data}|\alpha) = p(T_1=t_1,\ldots\ldots,T_n=t_n|\alpha) \tag{58}$$

we obtain for one time segment

$$\begin{aligned} p(\text{data}|\alpha) &= p(T_1=t_1|\alpha) \\ &= \int p(T_1=t_1|\lambda(1)=\ell)\cdot p(\lambda(1)=\ell|\alpha)d\ell \end{aligned} \tag{59}$$

For one time segment, the probability that the observed data has confidence level α is proportional to the probability that a specific failure time occurs given that a specific failure rate occurs and the probability that the specific failure rate has a certain confidence level. For n independent time segments

$$p(\text{data}|\alpha) = \prod_{i=1}^{n} \int p(T_i=t_i|\lambda(i)=\ell_i)(p(\lambda(i) = \ell_i|\alpha)d\ell_i \tag{60}$$

For continuous time

$$p_1(\alpha) = \prod_{i=1}^{n} \int f(t_i,\ell)\ g(\ell,i,\alpha)d\ell, \tag{61}$$

where $f(t_i,\ell)$ is the probability density function for failure rate and $g(\ell,i,\alpha)$ is the correction rule. Knowing $p_1(\alpha)$, the failure rate for time interval T_{n+1} can be calculated

$$p(\lambda(n+1) = \ell) = \int g(\ell,n+1,\alpha)\cdot p_1(\alpha)d\alpha, \tag{62}$$

and T_{n+1} can be calculated

$$p(T_{n+1}=t_{n+1}) = \int f(t_{n+1},\ell)\cdot p(\lambda(n+1) = \ell)d\ell \tag{63}$$

Equations (61) to (63) are the fundamental equations which define the Bayesian model. Basically the Bayesian model uses the predicted confidence level (61) to govern the selection of a failure rate for the n+1 time interval (62). From this failure rate, an estimation (63) of the next failure time can be calculated. Using the same development technique, the model was extended to predict time to n+1 failure from any arbitrary point between T_n and T_{n+1}.

The model becomes real when assignments are made to the distribution which governs the failure rates and correction rule. One assignment is that (64) was chosen because of the popular assumption that large system

$$f(t,\lambda) = \lambda e^{-\lambda t} \quad t > 0 \qquad \left.\begin{matrix} \\ \end{matrix}\right\} \lambda > 0 \qquad \text{Failure Rate} \tag{64}$$
$$= 0 \quad t < 0$$

$$g(\ell, i, \alpha) = \frac{\psi(i)(\psi(i)\ell)^{\alpha-1} e^{-\psi(i)\ell}}{\Gamma(\alpha)} \qquad \ell > 0 \qquad \text{Correction Rule} \tag{65}$$
$$= 0 \qquad \ell < 0$$

software behaves in a manner analoguous to large hardware systems. (65) was chosen because of its ability to represent many families of equations and because of its mathematical tractability. $\psi(i)$ is defined as the programmer's intention of improving a program.

From (61) to (65) calculation of an exact expression for $F(t_{n+1})$, the probability that $T_{n+1} < t_{n+1}$, and 100 α % confidence bound for $F(t_{n+1})$ can be made. The confidence bound is defined as follows

$$P\left[T_{n+1} < y_{\alpha}^{(n+1)}\right] = \alpha \tag{66}$$

where

$$y_{\alpha}^{(n+1)} = \psi(n+1)\left[\prod_{i=1}^{n} \frac{\psi(i)}{\psi(i) + t_i}\right]^{1-(1/1-\alpha)^{1/(n+1)}} - \psi(n+1) \tag{67}$$

$y_{\alpha}^{(n+1)}$ is infinite when the confidence α is 1 and 0 when α is 0.

Modeling the programmer's intention of improving a program $\psi(i)$ is a weakness of this model. One technique developed by the paper treats $\psi(i)$ as another parametric family $\psi(B,i)$. This technique rapidly degenerates into a numerical solution for estimating B and T_{n+1}. The numerical solution is implied to have a very heavy overhead.

$\psi(i)$ can be empirically estimated, and Littlewood and Verrall [12, 13] provide a technique to measure the quality of any $\psi(i)$. The technique is based on the idea that at each failing time a confidence bound can be obtained, conditional upon the observed data. If the parametric family or empirical estimate of $\psi(i)$ is representative of the programmer's intention of improving the program then the technique shows that the portion of the data which meets the bounding criteria (calculated by (67)) will be proportional to the specific confident value α used to define the bounding time for the next interval. The more representative $\psi(i)$ is, the more that the proportion approaches an equality.

Research [14] is being conducted to more precisely define the correction rule. In the Bayesian model, corrections are performed instantaneously with regard to levels of correction personnel, correction times, and correction strategies. This research is directed toward dealing with these limitations.

The Bayesian model has not been validated on real data. Real data was used to demonstrate that the modeling process was capable of generating realistic figures for various models of $\psi(i)$. This data is summarized in Table 4.

Table 4 Bayesian Model Validation Data

i	1	2	3	4	5	6	7	8	9	10	11	12
t_i (DAYS)	9	12	4	20	72	94	9	36	37	70	6	115

$\psi(i)$	MEDIAN of T_{13}
$B_0 + B_1 i^2$	101.5
$\exp(B_0 + B_1 i)$	132.9
$B_0 i + B_1 i^2$	76.7
$B_0 + B_1 i$	61.8
$B_0 + B_1 i^3$	142.7

3.9 Markov Model

This modeling technique [15,16] models large system software as a discrete state, continuous time Markov process. The major assumptions made are that the program is very large (10^5 words), can run for fixed periods of time, the number of failures is fixed, and failure detection and correction occur alternately and sequentially before operation resumes. This model can be concisely illustrated as a diagram shown in Figure 11 having n nodes, where n is the number of failures. At t_0 the model has n failures. As the software is exercised, failures will surface at a rate λ. The failure rate can be constant or can vary with the number of failures that remain in the software. Once a failure surfaces, a transition is made to a correction state, where it is repaired by a perfect repair person whose rate is μ. The

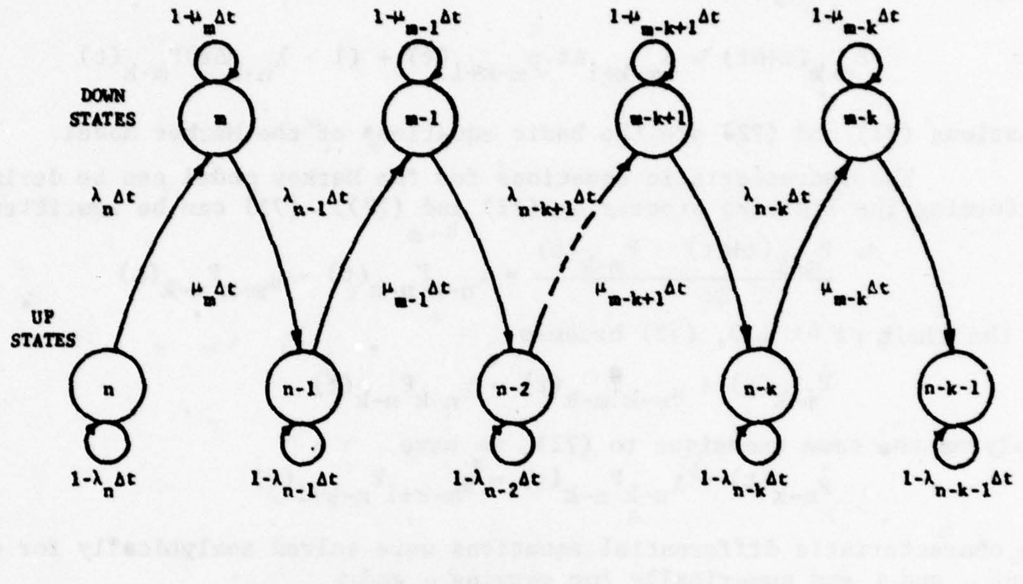


Figure 11 Markov Modeling Process

repair rate has the same characteristics as the failure rate. States in which correction is taking place are called down-states. The probability of remaining up-state during the next instant of time for the initial state (n) is given by

$$P_n(t+\Delta t) = (1 - \lambda_n\Delta t)P_n(t) \qquad (68)$$

The probability of discovering a failure when in state n at t and making a transition to state P_m so that at t+Δt correction is taking place as given by

$$P_m(t+\Delta t) = \lambda_n\Delta t P_n \qquad (69)$$

The probability of remaining in state P_m for the next instant of time is

$$P_m(t+\Delta t) = 1 - \mu_m\Delta t P_m(t) \qquad (70)$$

The probability of being in the down state which repairs the kth failure is the sum of the probability that k errors are discovered and the probability that k errors were discovered but not repaired, i.e.,

$$P_{m-k}(t+\Delta t) = \lambda_{n-k}\Delta t \cdot P_{n-k}(t) + (1-\mu_{m-k})\Delta t \cdot P_{m-k}(t) \qquad (71)$$

Similarly, the probability of being in the upstate and awaiting the arrival of the kth failure is

$$P_{n-k}(t+\Delta t) = \mu_{m-k+1}\Delta t \cdot P_{m-k+1}(t) + (1 - \lambda_{n-k}\Delta t)P_{m-k}(t) \qquad (72)$$

Equations (71) and (72) are the basic equations of the Markov model.

The characteristic equations for the Markov model can be derived by performing the limiting process on (71) and (72). (71) can be rewritten as

$$\frac{P_{m-k}(t+\Delta t) - P_{m-k}(t)}{\Delta t} = \lambda_{n-k}P_{n-k}(t) - \mu_{m-k}P_{m-k}(t) \qquad (73)$$

As the limit of $\Delta t \to 0$, (73) becomes

$$\dot{P}_{m-k}(t) + \mu_{m-k}P_{m-k}(t) = \lambda_{n-k}P_{n-k}(t) \qquad (74)$$

Applying the same technique to (72), we have

$$\dot{P}_{n-k}(t) + \lambda_{n-k}P_{n-k}(t) = \mu_{m-k+1}P_{m-k+1}(t) \qquad (75)$$

The characteristic differential equations were solved analytically for constant μ and λ and numerically for varying μ and λ.

The primary parameters <u>measured</u> by this model are availability and nonavailability of the software modeled by this process. These parameters are only predicted on an average basis in practice because the assumption that each error is corrected before a new error is discovered does not happen for every error discovered.

Availability and non-availability are calculated by summing up up-states and down-states at various points in time.

$$\text{AVAILABILITY} \qquad A(t) = \sum_{k=0}^{k_{max}} P_{n-k}(t) \qquad (76)$$

$$\text{NON-AVAILABILITY} \qquad B(t) = \sum_{k=0}^{k_{max}} P_{m-k}(t) \qquad (77)$$

The main strength of this model is its simplicity. From its availability estimates, better resource allocations can be made. The primary weakness is the fact that it has not been validated. An attempt to <u>validate</u> the model was made by Sukert [19] but it produced inconsistency. Another weakness is the assumption used by this model that the number of failures is constant. It assumes no error generation during correction, and it contains no classification of failures.

3.10 <u>Shooman's Improved Model</u>

Shooman [4] has identified three types of repair personnel. The first type was the perfect repair personnel. Software debugging using the perfect repair person results in a correction process shown in Figure 2. Software debugging using this class of personnel to perform debugging obviously improves the software with time. The second type of repair person spawns errors at the rate equal to which he corrects errors. This process is depicted by Figure 12. Software debugging using this class of personnel does

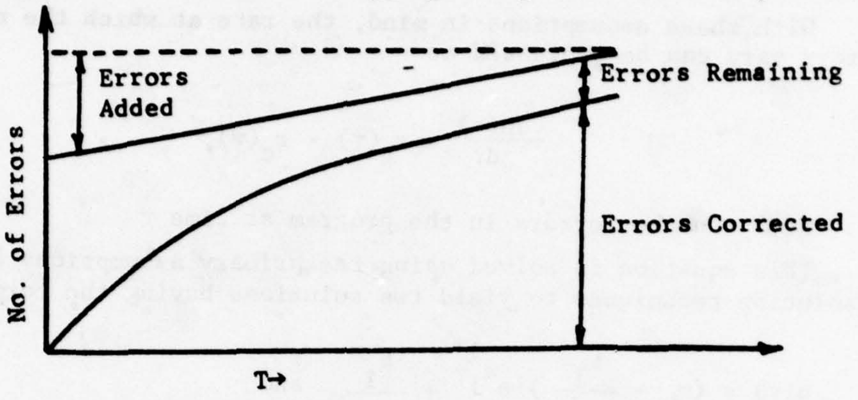


Figure 12 Debugging Process (Imperfect Repair Person - II)

not necessarily improve the software with time. The third class of repair person is one who causes more failures to be induced into the software than he/she corrects. The process for this individual is depicted by Figure 13.

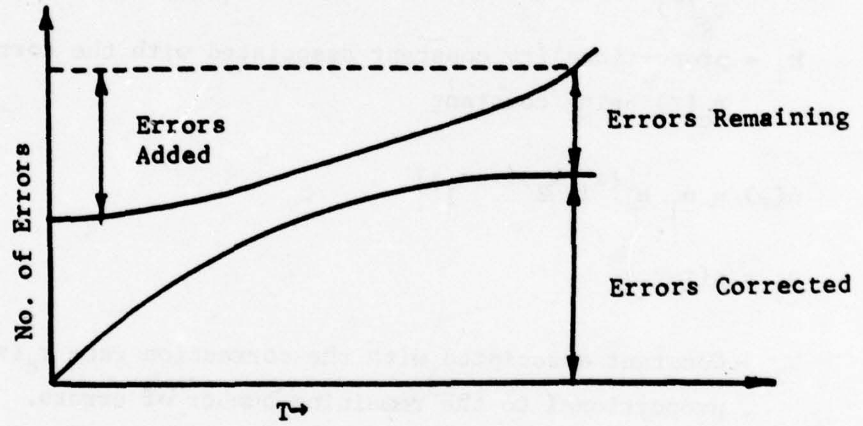


Figure 13 Debugging Process (Imperfect Repair Person - III)

Shooman's first model [1-5] treated the case where correction is performed by a perfect repair person. His improved model [18] includes treatment of imperfect repair personnel.

The primary assumptions made by the improved model are

1) the error detection rate $r_d(\tau)$ is constant.

2) the error correction rate $r_c(\tau)$ is constant, and sometimes proportional to the remaining number of errors.

3) the spawning rate of new errors $r_g(\tau)$ is proportional to the

error detection rate and remaining number of errors.

Assumption 1 is common to many models previously examined. Assumption 2 is a popular view shared by many program managers. Assumption 3 makes intuitive sense.

With these assumptions in mind, the rate at which the number of program errors vary can be expressed as

$$\frac{dn(\tau)}{d\tau} = r_g(\tau) - r_c(\tau), \tag{78}$$

where $n(\tau)$ = number errors in the program at time τ

This equation is solved using the primary assumptions and conventional solution techniques to yield two solutions having the form of

$$n(\tau) = (n_0 - \frac{k_1}{a_1}) e^{a_1\tau} + \frac{k_1}{a_1} \quad \text{and} \tag{79}$$

$n_0 = n(\tau=0)$ = number of errors in the program at time 0

a_1 = proportionality constant associated with the spawning rate $r_g(\tau)$

k_1 = proportionality constant associated with the correction rate $r_c(\tau)$ being constant

$$n(\tau) = n_1 e^{[(a_1-k_2)(\tau-\tau_1)]} \tag{80}$$

where $n_1 = n(\tau=\tau_1)$

k_2 = Constant associated with the correction rate $r_c(\tau)$ being proportional to the remaining number of errors.

(79) represents the solution for the model when the constant error correction rate is valid. Similarly (80) represents the solution for the model when the proportional error correction rate is valid. Figure 14 better illustrates the interaction between these two equations.

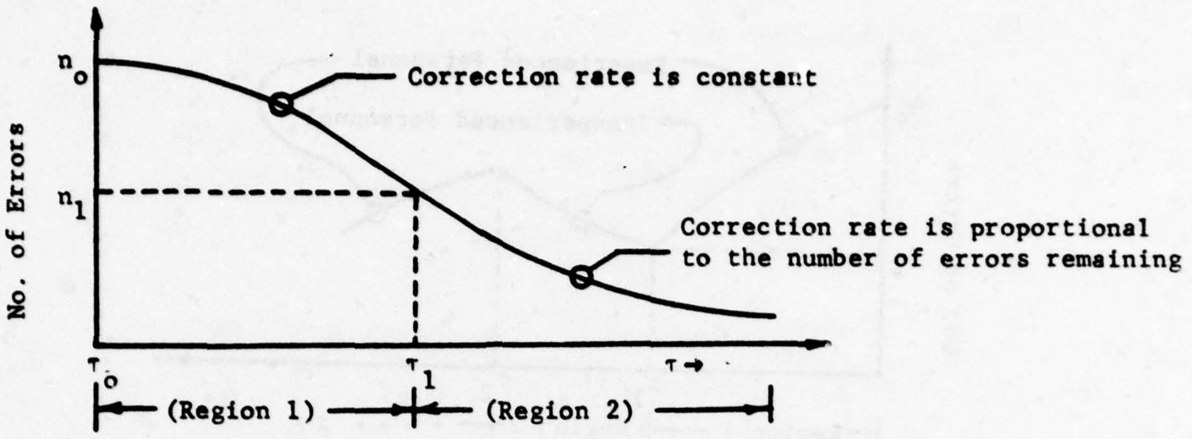


Figure 14 Normal Error Debugging

In region one when the software is young; debugging activity is constrained by the number of repair personnel available to correct detected failures. Hence, correction activity is a constant. In region two, the software has matured to the point where correction personnel are not always busy. Hence, correction is proportional to the number of remaining errors and the detection rate.

The model generalizes to account for various possible correction/generation activity which might occur. For example in region one, correction personnel could be inexperienced and cause the software to become unstable (see Fig. 15).

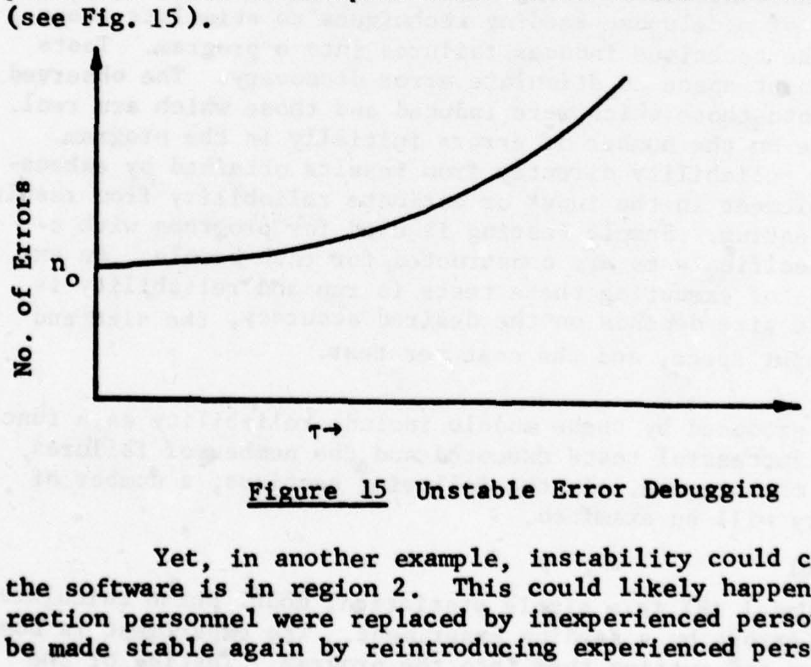


Figure 15 Unstable Error Debugging

Yet, in another example, instability could creep in shortly after the software is in region 2. This could likely happen if experienced correction personnel were replaced by inexperienced personnel. The systen could be made stable again by reintroducing experienced personnel.

This situation is depicted by Figure 16.

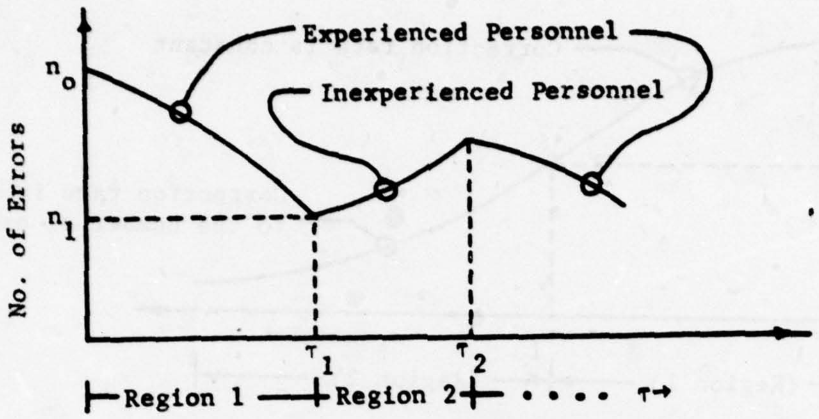


Figure 16 Quasi-Stable Error Debugging

Shooman's Improved Model is more realistic in its attempts to model the software debugging process. Unfortunately, lack of data has prevented validation and refinement.

4. TIME INDEPENDENT MODELS

Time independent models calculate reliability from observed results of experiments conducted on elements of the program's input space. Experiments are time independent and constructed using some a priori knowledge of the input space. A couple of models use seeding techniques to stimulate observation of failures. The technique induces failures into a program. Tests are conducted on the input space to stimulate error discovery. The observed errors are separated into those which were induced and those which are real. Estimations can be made on the number of errors initially in the program. Other models calculate reliability directly from results obtained by exhaustively testing every element in the input or estimate reliability from results obtained from sample testing. Sample testing is used for programs with a large input space. Specific tests are constructed for each sample. An experiment which consists of executing these tests is run and reliability is calculated. The sample size depends on the desired accuracy, the size and distribution of the input space, and the cost per test.

Figures of merit produced by these models include reliability as a function of the number of successful tests executed and the number of failures initially inherent in the program. In the following sections, a number of models in this category will be examined.

4.1 Mills' Model

The Mills' Model (8) is a simple statistical model which calculates the number of residue errors by a seeding experiment. The experiment is composed of tagging errors and putting them into the program. Testing of the software starts. After a period of time, an examination is made of the errors recorded to date. The errors are classified into two categories: 1) seeded and 2) indigenous.

A prediction of the number of indigenous errors [X] can be made by maximum likelihood estimator [26]

$$X = \left\lceil \frac{Y\,U}{V} \right\rceil , \tag{81}$$

where Y = number of induced failures (seeded)

U = number of indigenous failures found

V = number of induced errors found to date

The main assumptions made by this model are

1) $P(E_i)$ = Constant (independent of the number of failures previously found)

2) One failure can be found at a time

To test the validity of this model, 20 failures were seeded into a program with 10 residual failures. After debugging 12 failures, 8 were predicted to be indigenous.

A weakness of seeding techniques is accuracy. Intuitively better accuracy is obtained when the number of seeded failures is much greater than the number of indigenous failures. This effect will be addressed in the next model.

4.2 *Lipow's Model*

Lipow [27] extended Mills' model to include the probability of finding a failure. Basically, his experiment consisted of conducting N statistically independent tests (where N is less than or equal to the number of seeded failures) such that each test results in finding an error of either kind (success) or finding no error. The probability of success for such an experiment can be expressed as

$$P_N(X_I,X_S;\, q,n_I,n_S) = \binom{N}{X_I + X_S} q^{X_I + X_S} (1-q)^{N-X_I-X_S}\, \frac{\binom{n_I}{X_I}\binom{n_S}{X_S}}{\binom{n_I + n_S}{X_I + X_S}} \tag{82}$$

where q = probability of finding a failure

n_I = number of indigenous failures

n_S = number of seeded failures

N = number of tests (statistically independent)

X_I = number of indigenous failures found via experiment

X_S = number of seeded failures found via experiment

The assumptions made by this model are the same as those made by the Mills' model. Maximum likelihood estimation techniques are used to develop predictors for estimating q and n_I. Their expressions are

$$\hat{q} = \frac{X_I + X_S}{N} \tag{83}$$

$$\hat{n}_I = \left[\frac{X_I}{X_S} n_S\right] \tag{84}$$

As previously mentioned in the Mills' discussion, one of the weaknesses of the seeding technique is accuracy. How accurate are $\hat{q}$ and $\hat{n}_I$ for a given number of seeded failures (n_S) and tests (N)? Lipow was able to put a handle on this question by deriving the following expressions for expected value and deviation

$$E(\hat{n}_I | r) = \sum_k kP(\hat{n}_I = k | r) \qquad r = X_I + X_S \tag{85}$$

$$\sigma(\hat{n}_I | r) = \left[\sum_k (k - E(\hat{n}_I | r))^2 P(\hat{n}_I = k | r)\right]^{1/2} \tag{86}$$

where

$$P(\hat{n}_I = k | r) = \sum_{X_S \in K} \frac{\binom{n_I}{r - X_S}\binom{n_S}{X_S}}{\binom{n_I + n_S}{r}} \tag{87}$$

where K is the set of X_S such that $X_S \geq 1$ and $\hat{n}_I = k$ or $X_S = 0$ and $rn_S = k$. (85) defines the expected number of indigenous failures given r observed successes. (86) defines how much $\hat{n}_I$ is expected to deviate from the norm. Lipow performed a sample data analysis using the above equations. The analysis showed that accuracy tended to improve by increasing n_S and by making N equal to n_S.

This model is an application of classical statistics. It was proposed as a technique to estimate residual failures in software. No validation of this model was given or referenced in this report. The author cited future research areas to generalize the model. One generalization was to incorporate a technique to decrease the probability of finding an error when errors are found. Another was to develop some weighting scheme so that the probability of finding an indigenous failure would not be equal to the probability of finding a seeded error. To optimize testing time, one would want to weight indigenous failures more heavily. Finally some techniques should be developed to account for multiple failures.

4.3 Rudner's Model

Rudner [28] also extended Mills'model. Using Mills'assumptions, Rudner recognized that the seeding/tagging procedure could be developed into the classical statistical experiment of sampling without replacement. In her debugging experiment, a program tester runs a set of s tests on a program with N (including those seeded) errors. Each test is capable of discovering one error. Upon completion of test, the errors which were discovered contain seeded errors and errors indigenous to the software. The probability that the number of seeded errors equal c after the testing is complete can be computed by

$$P(c|s,t,N) = \frac{s!t!(N-s)!(N-t)!}{N!c!(s-c)!(t-c)!(N-s-t+c)!} \tag{88}$$

where c = number of seeded errors found by debugging

s = number of errors discovered

t = number of seeded errors

N = number of errors in the software when testing begins (includes seeded errors)

From this model, the number of errors in the software (N) when testing begins can be estimated two ways:

$$N_1 = \left[\frac{s+t}{c}\right] \quad \text{(Estimator 1)}$$

$$N_2 = \left[\frac{(s+1)\ (t+1)}{c+1} - 1\right] \quad \text{(Estimator 2)} \tag{89}$$

Estimator 1 is derived using a maximum likelihood technique. N_1 can be calculated using data obtained from the testing experiment. A disadvantage associated with this estimator is that it predicts estimates that are normally higher than N. Another disadvantage is that estimator 1 is invalid if the number of seeded errors found by debugging is 0. Estimator 2 was derived to compensate for estimator 1.

Rudner relaxes the Mills' assumption that all errors are equally probable and defines another model that handles errors of variable difficulty. This model makes the following assumptions:

1) All bugs can be assigned at sight to categories based on difficulty of discovery.
2) Within each catagory, errors are subject to random discovery with equal probability.

The previous model is then applied to each category individually and their estimates are summed to define the program estimate. This model requires k seeding/tagging reliability tests where k is the number of levels of difficulty. By making a third assumption, it is possible to compute an estimate of the number of total errors using one seeding/tagging experiment. The assumption is that the distribution ratio of program errors by category is known. Errors are seeded into the program in proportion to the ratio for each category. The number of estimated program errors is calculated using estimators 1 or 2.

These models were proposed as a way of estimating the number of error in software when testing. The basic approach taken was to make the assumptions that

1) All errors or errors within a category have equal probability of discovery, and

2) Errors and discovery occur one at a time.

The seeding/tagging procedure was then recognized to be a classical statistic model. The practicality of these models is unknown since they have not been validated.

4.4 Nelson's Model

Nelson [29-31] published several statistical models to measure reliability. The primary assumption made by these models is that a program bug causes an execution failure. Program reliability is then measured as

$$R = 1 - \frac{n}{N} \tag{90}$$

where n = number of inputs with execution failures

N = total number of inputs

Reliability measured in this fashion can be estimated by

$$R = 1 - \frac{n_e}{n}, \tag{91}$$

where n = number of random samples of inputs

n_e = number of sample inputs which provide an output having a failure.

Nelson improves these models by observing that all inputs are not equally executed. By making the following assignments

$$p_i = \text{Prob (input } i \text{ is used during operational use)} \tag{92}$$

$$X_i = \left\{ \begin{array}{ll} 1 & \text{Correct output for input } i \\ 0 & \text{Otherwise} \end{array} \right\} \tag{93}$$

Reliability is defined by

$$R = \sum_{i=1}^{N} p_i X_i \tag{94}$$

Observing that a program can only make a finite number of unique computations yields an improvement in another direction. Let the set of finite computations be represented as

$$E = \{E_i : i = 1, 2, \ldots\ldots N\} \tag{95}$$

Associated with each unique computation is an execution variable y_i having the form:

$$y_i = \begin{cases} 0 \text{ - Run is O.K.} \\ 1 \text{ - Run has execution failure} \end{cases} \tag{96}$$

By letting p_{ij} be the probability that computation i is chosen for run j in a sequence of runs and P_j be the probability that run j results in an execution failure then P_j can be calculated by

$$P_j = \sum_{i=1}^{N} p_{ij} y_i \tag{97}$$

The program's reliability (for n runs) can be expressed as

$$R(n) = (1-P_1)(1-P_2)\ldots(1-P_n) \tag{98}$$

$$= \pi_{j=1}^{n} (1-P_j)$$

R(n) can be expressed in its exponential form

$$R(n) = \mathrm{EXP}(-\sum_{j=1}^{n} \ln(1-P_j)) \tag{99}$$

Assuming that the probability that run j resulting in an execution failure is much less than unity, then

$$R(n) = \mathrm{EXP}(-\sum_{j=1}^{n} P_j) \tag{100}$$

If all P_j equal P, then

$$R(n) = \mathrm{EXP}(-Pn) \tag{101}$$

An important relationship which relates time independent (statistical) reliability R(n) to time dependent reliability R(t) was defined during this development. The relationship is evident in the following transformation.

$$R(n) = \mathrm{EXP}(-\sum_{j=1}^{n} \ln(1-P_j)) \tag{102}$$

letting

$$\ln(1-P_j) = Z(t_j)\Delta t_j \tag{103}$$

where Δt_j is the time for run j and $Z(t_j)$ is a failure rate associated with run j.

$$R(t_n) = \text{EXP}(-\sum_{j=1}^{n} Z(t_j)\Delta t_j) \tag{104}$$

If $\Delta t_j = \Delta t$ and $Z(t_j) = Z(t) \forall\, i < n$

then
$$R(t_n) = \text{EXP}(-\sum_{j=1}^{n} Z(t)\Delta t) \tag{105}$$

Finally if $t \to 0$

$$R(t) = \text{EXP}(-\int_{o}^{t} Z(t)dt) \tag{106}$$

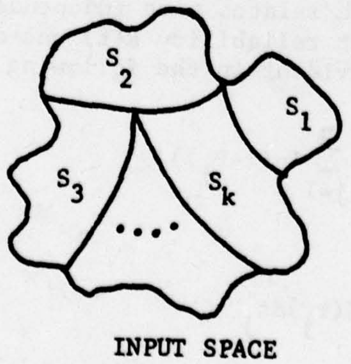
which is the basic form of time dependent models (see (2)).

These models were enhanced and unified into a mathematical theory of software reliability in a later work [32]. In this later work a measurement technique based on (91) was developed to measure R(n). The technique was validated on two programs coded from the same specification by two individual programmers. Reliability for program 1 was estimated to be .997. Reliability for progcam 2 was estimated to be .965. The program having the higher reliability had a simpler structure.

4.5 Brown-Lipow Model

Brown-Lipow developed a model [33] on the thesis that if all possible input data used by a program could be somehow partitioned, then some estimate could be made on the program's reliability. They developed this thesis after observing that all elements which make up a program's input data were not uniformly exercised when the program was operational. By letting all data elements define an input space, Brown-Lipow developed two techniques to partition the input space. 1) The 'S' partition divides the input space by magnitude. 2) The 'G' partition divides the input space by the ability to execute a certain class of control paths. Two reliability estimators, observable and operational, were developed using the ability to partition the input space.

The 'S' partition is defined and depicted in Figure 17 . An example



$$E = S_1 \cup S_2 \cup S_3 \ldots \cup S_k$$
$$\emptyset = S_1 \cap S_2 \cap S_3 \ldots \cap S_k$$

INPUT SPACE

Figure 17 'S' Partition Over Input Space of how to form a couple of partitions in E would be to let S_1 be the set of all inputs E_i that are less than X in magnitude. Let S_2 be the set of all inputs that are greater than or equal to X in magnitude. More formally, let

$$S_1 \overset{\Delta}{=} E_i = (X_1, X_2, X_3, \ldots, X_j, \ldots, X_n)_i \quad X_j < X \tag{107}$$

$$S_2 \overset{\Delta}{=} E_i = (X_{n+1}, \ldots)_i \quad X_j \geq X \tag{108}$$

Now if one carefully chooses X, one could assign the following probabilities $P(S_1) = P(X_j < X) = .8$ and $P(S_2) = P(X_j \geq X) = .2$.

Another approach at partitioning the input space involves defining the program's logic path space (L).

$$L = L_1, L_2, \ldots, L_i, \ldots, L_m \tag{109}$$

A logic path L_i is a sequence of adjacent segments, beginning at an entry segment and proceeding by logical transfers to an exit segment. $L_1, L_2, \ldots, L_m$ can be partitioned under the following rules:

$$L = L_1' \cup L_2' \cup L_3' \cup \ldots \cup L_k' \; \ldots \tag{110}$$

$$\emptyset = L_1' \cap L_2' \cap L_3' \cap \ldots \cap L_k' \; \ldots \tag{111}$$

L_k' is a subspace which shares a common characteristic among two or more L_i. For example, L_1' could be the set of all L_i which contain a specific transfer. L_2' would be the set of all L_i which does not contain the transfer. Now, let $P(L_k')$ be the distribution which estimates the relative frequency of occurrence of an event associated with each L_k'. For example if L_1' represented a characteristic common to half of the logic paths, then assuming that all logic paths are uniformly exercised, one could estimate $P(L_1')$ to equal .5.

The <u>'G' Partition</u> is defined as

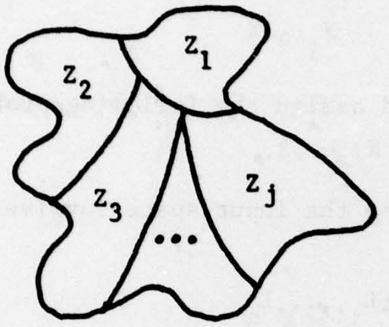
$$G = G_1 \cup G_2 \cup G_3 \cup G_4 \cup \ldots \cup G_k \tag{112}$$

$$\emptyset = G_1 \cap G_2 \cap G_3 \cap G_4 \cap \ldots \cap G_k \tag{113}$$

where G_k contain all the elements in the input space which cause execution of one of the paths in L_k'. The operational profile probability distribution

$P(G_k)$ is identical to $P(L_k')$.

Given some valid technique(s) (e.g., 'S' and 'G' partitioning techniques are valid) which partitions the input space as defined and illustrated by Figure 18 , then Z_j can be further partitioned into a subset Z_j' which yields a correct execution



$$Z = Z_1 \cup Z_2 \cup Z_3 \ldots \cup Z_j$$

$$\emptyset = Z_1 \cap Z_2 \cap Z_3 \ldots \cap Z_j$$

<u>Figure 18</u> 'Z' Partition Over Input Space

of the input space and a subset Z_j'' which yields an incorrect execution of the input space. Formally expressed

$$Z_j = Z_j' \cup Z_j'' \quad \text{and} \tag{114}$$

$$P(Z_j) = P(Z_j') + (Z_j'') \tag{115}$$

Reliability can be expressed as the probability that all elements required for execution will be contained in Z_j' or

$$R = \sum_j P(Z_j') = 1 - \sum_j P(Z_j'') \tag{116}$$

Two reliability estimators can be easily computed:

<u>Estimator #1</u> - Operational Usage Reliability

$$R_1 = 1 - \left[\frac{f_1}{n_1} P(Z_1) + \frac{f_2}{n_2} P(Z_2) + \ldots + \frac{f_j}{n_j} P(Z_j) \right] \tag{117}$$

where f_j - number of failures in n_j tests using data elements in Z_j

$P(Z_j)$ - operational profile probability (a measure on how data elements of Z_j affect reliability)

If n executions of the program are identically proportional to the operation profile distribution then reliability can be estimated by

Estimator #2 - Operational Usage Reliability

$$R_2 = 1 - \left[\frac{f_1}{nP(Z_1)} \cdot P(Z_1) + \frac{f_2}{nP(Z_2)} \cdot P(Z_2) + \ldots \right] \tag{118}$$

$$= 1 - \frac{1}{n} \sum_j f_j$$

A Chi-Square Technique is used to monitor the validity of assumption (112).

$$\chi^2 = \sum_j \frac{(n_j - nP(Z_j))^2}{nP(Z_j)} \tag{119}$$

Brown and Lipow validated their model using the 'G' partition to classify the input space for a small program. From their example one can deduce that classifying the input space is a nontrivial task. Z_j', Z_j'', $P(Z_j')$, and $P(Z_j'')$ are hard to determine short of using manual-exhaustive techniques. For large programs, this classification job could become unrealistic.

The basic concepts of the Brown-Lipow model have been incorporated into a later study[32] as an extension to theory defined by Nelson.[29-31] An automated test tool, PACE, is used to aid in defining the 'G' partition.

5. SUMMARY

Two distinct classes of models have been examined in this report. The time dependent class treats programming errors as an inherent program property. Assumptions are made about this property and used to model error behavior as a function of time. Many models have been proposed, built, and partially validated. The time independent class treats errors in a slightly different manner. Errors are handled in a statistical manner. Tests which specifically check for error conditions are carefully defined using a priori knowledge of the program. These tests form an experiment which is conducted on the software system under test. Experimental results are used to calculate figures of merit. Several models have been proposed, some have been built and partially validated.

The purpose of modeling is to supply meaningful figures of merit to software managers to aid them in their endeavor to better manage resources. Figures of merit produced by the models in this survey are: 1) the number of failures inherent in a program at anytime, 2) reliability as a function of time, 3) reliability as a function of the number of successful tests, and 4) mean time to next failure. Associated with each figure of merit is a degree of accuracy which is dependent on the validity of a specific model, the observed data, and the quality and number of tests.

The usefulness of these figures of merit is still questionable. More experience is required before their full potential can be realized. A potential use for the number of failures inherent in a program would be to supply a manager the knowledge of how fast his test team is approaching to the point in time where testing becomes not too cost effective. The number of errors left in the program when it becomes operational can be used to aid a manager in selecting the number of people required for program maintenance. Reliability figures of merit show possibility of being used for comparison purpose. Two programs can be compared by their reliability figures. When the mean time to next failure and the information on the inherent number of errors are available, management can use the information to determine whether the number of software maintenance people required will be large for a short duration of time or small for a long duration of time.

5.1 Time Dependent Models

Time dependent models treat software as a black box system that will fail at some time in some deterministic manner. They assume that elements of the program's input space are randomly selected and representative of the program's operational environment. Failures which occur are independent and proportional with the current error content. Time dependent models can be categorized into four distinct groups. 1) The Shooman, Execution, and Weibull models have continuous failure rates and are capable of estimating the residue failure content, reliability, and MTTF. These models use the CPU time as a time base. 2) The De-Eutrophication styled models have discontinuous failure rates. They are capable of estimating failure contents, reliability, and MTTF. The time base for these models is CPU or calendar except for the Geometric-Poisson. Its time periods are fixed intervals measured in months. 3) The Markov model is distinct in the sense that it models software failure as a Markov process. Because of this, it most naturally measures program availability and non-availability. 4) The Bayesian Model measures the MTTF and attempts to model the behavior of the repair personnel.

Several evaluation studies [10,19,21] have been made on group 1 and 2 models. Results of these studies are somewhat inconclusive because of inconsistencies (e.g. one study is capable of using a model that some other study cannot.) Model accuracy, including the basic ability to even use a specific model, seems to be very data dependent. Where models have been successfully applied, results are reasonable. Of group 1 and 2 models, the most sophisticated and probably the most accurate is the Execution Time model.

Very little validation effort has been published on group 3 and 4 models. Both groups are built on sound mathematical theory. The Bayesian model is probably the more realistic and complex. It attempts to model error spawning by the imperfect repair process by dynamically adjusting the failure rate. Only two other models account for the effects of error spawning. One accounts for error spawning on a static (average) basis. This is primarily due to the constant error constraint imposed on group 1 and 2 models. The other accounts for error spawning using a birth-death process.

The Markov model has the versatility of controlling the failure and repair rate as a function of the number of failures which currently exist. The result of this capability has not been fully explored.

In general, more evaluation studies are required to determine possible model refinements to produce consistent results. Evaluation studies should analyze the validity of basic assumptions made by time dependent modules. For example, consider the "Equal Exposure" assumption, which says that the exposure rate is constant (i.e. $E(t) = \frac{E_t}{I_t}$). Ideally one wants to design tests to conform to that assumption, but it is often the case that tests do not conform. Hence, inaccuracy creeps into model results due to partial violation of a basic assumption. In this particular situation, inaccuracy could be compensated by using some combination of path and node frequencies measurement to define e(u) which in turn can be used to calculate the exposure rate $[E(t) = \int_0^t e(u)du]$. If E(t) is not uniform, then the distribution of e(u) can be used to improve the accuracy of the model.

5.2 Time Independent Models

Time independent (statistical) models treat software as a system which performs transformations on an input space. If a transformation is performed correctly, then the experiment which provoked the transformation is a success. To add accuracy to the results, most models attempt to predict some information about the program's input space. This prediction can be as simple as assuming a uniform distribution over the input space and then assigning a probability that an element from the space causes an error. Predictions can be more sophisticated (e.g. partitioning the input space deterministic on the program's internal characteristics and formulating tests which specifically exercise a certain portion of the input space.)

Time independent models appear to provide a more natural mechanism (but not necessarily less complex) for predicting reliability. Given the knowledge of a program's input space and its operational profile distribution, realistic predictions can be made about programs of any sizes. The complexity of this process comes from partitioning the input space and measuring its operational profile. A unified semiautomatic methodology [32] using the concepts put forth during discussion of the TRW models [29,31,33] has been successfully demonstrated.

6. FUTURE RESEARCH

As we can see from the previous description and discussion, although there have been substantial results in software reliability modeling, much work still needs to be done in order to have significant practical use. The main problems with the current models are the assumptions used and the lack of validation of these models. Most of the existing models (time dependent class of models) employ the "black box" approach or some (time independent class of models) use the information of the program structure to partition the input space. For the time dependent class of models, there is not usually any sufficient justification for the reality of the assumptions used by these models and when there is no meaningful way to validate the models, little confidence can be established on the use of these models. For time independent class of models, the main difficulty is the complicated process involved in partitioning the program input space.

Another direction of research in this area is to develop new figures of merit that can be defined to aid management in their decision making processes for software maintenance. Current models do not account for the time required for error correction. Many models assume that repair is performed instantaneously by a perfect repairperson. Two models [12,17] admit that the repairperson is imperfect, though his actions are instantaneous. Another model [15] associates a repair rate with each failure; and in this case repairs are performed by a perfect repairperson. In reality when an error surfaces, a written record may be made, and placed on a stack of similar records. The size of the stack is dependent on the resources (correction personnel and computer time) available for correction and the rate which errors surface. If failures are discovered faster than they are fixed, the stack size gets large. Management might try to counter a growing stack by supplying more personnel to perform corrections. If this is the case, the stack will probably decrease. Management would then reassign personnel to other tasks to keep them busy. Other situations governed by stack size can be cited which causes management to shift personnel.

To aide management in resource allocation, three new figures of merit could be modeled. The first is the projected stack size. With this figure of merit, a manager could obtain an estimate of the required personnel. This knowledge would facilitate personnel movement since individuals appreciate advance notice of their job status. The second figure of merit could speculate how long error correction takes. Error correction is a function of an individual's familiarity with the software under test; hence, the first correction may not work thus requiring a second try. This process requires time. With knowledge of the average repair time, managers could speculate whether they are efficiently managing their resources. The third figure of merit is the sensitivity of the improvement of software reliability with respect to the reliability improvement of individual modules. This has been investigated using an entirely different model, called a user-oriented software reliability model [35], which is based on the program structure and the user profile. However, additional work still needs to be done in order to make it useful.

7. REFERENCES

1. M. L. Shooman, "Probabilistic Models for Software Reliability Predictions," Statistical Computer Performing Evaluation, edited by Walter Freiberger, Academic Press 1972, pp. 485-502.

2. M. L. Shooman, "Probabilistic Models for Software Reliability Prediction," 1972 International Symposium on Fault-Tolerant Computing, Newton, Mass. June 19-21, pp. 211-215.

3. J. C. Dickson, J. L. Hesse, A. C. Kuentz, M. L. Shooman, "Qualitative Analysis of Software Reliability," Proc. of the Annual Reliability and Maintainability Symposium, San Francisco, 1972, pp. 148-157.

4. M. L. Shooman, "Operational Testing and Software Reliability During Program Development," Record of IEEE Symposium Computer Software Reliability, New York City, 1973, pp. 51-57.

5. M. L. Shooman, "Software Reliability: Measurements and Models," 1975 Annual Reliability and Maintainability Symposium, Washington, D. C., pp. 485-490.

6. Z. Jelinski and P. B. Moranda, "Software Reliability Research," Statistical Computer Performance Evaluation, edited by Walter Freiberger, Academic Press 1972, pp. 465-485.

7. Z. Jelinski and P. B. Moranda, "Application of a Probability-Based Model to a Code Reading Experiment," Proc. 1973 IEEE Symposium on Computer Software Reliability, pp. 78-81.

8. P. B. Moranda, "Predictions of Software Reliability During Debugging," 1975 Proc. of the Annual Reliability and Maintainability Symposium, Washington, D. C., Jan. 1975, pp. 327-332.

9. R. W. Wolverton and G. J. Schick, "Assessment of Software Reliability," TRW Systems Group, Report No. TRW-SS-72-04, Sept. 1972.

10. W. L. Wagoner, "The Final Report on a Software Reliability Measurement Study," The Aerospace Corp., Report No. TOR-0074(4112)-1, Aug. 15, 1973.

11. J. D. Musa, "A Theory of Software Reliability and Its Application," IEEE Trans. on Software Engineering, Vol. SE-1, No. 3, Sept. 1975, pp. 312-327.

12. B. Littlewood and J. L. Verrall, "A Bayesian Reliability Growth Model for Computer Software," 1973 IEEE Symposium Computer Software Reliability, New York City, pp. 70-77.

13. B. Littlewood and J. L. Verrall, "A Bayesian Reliability Growth Model for Computer Software," Journal of the Royal Statistical Society, Series C, Applied Statistics, 1973, pp. 332-346.

14. A. L. Goel, "Summary of Technical Progress on Bayesian Software Prediction Models," RADC-TR-112, March 1977, A039022.

15. A. K. Trivedi and M. L. Shooman, "A Many-State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters," 1975 International Conference on Reliable Software, pp. 208-220.

16. M. L. Shooman and A. K. Trivedi, "A Many-State Markov Model for Computer Software Performance Parameters," IEEE Trans. on Reliability, Vol. R-25, No. 2, 1976, pp. 66-67.

17. M. L. Shooman and S. Natarajan, "Effect of Manpower Deployment and Bug Generation on Software Error Models," RADC-TR-76-40, Jan. 1977.

18. M. L. Shooman, "Probabilistic Reliability: An Engineering Approach," McGraw-Hill Book Co., New York, NY, 1968.

19. A. N. Sukert, "A Software Reliability Modeling Study," RADC-TR-76-247, Aug. 1976, A030437.

20. A. N. Sukert, "An Investigation of Software Reliability Models," Proc. 1977 Annual Reliability and Maintainability Symposium, pp. 478-484.

21. L. G. Stucki, "Final Report: A Methodology for Producing Reliable Software - Vol. 1," McDonnel Douglas Astronautics Co., March 1976.

22. I. Miyamoto, "Software Reliability in Online Real Time Environment," Proc. 1975 International Reliable Software, Los Angeles, CA, April, 1975, p. 198.

23. F. Akiyama, "An Example of Software System Debugging," IFIP Congress 1971, Ljubljana, Yugoslavia, August 1971.

24. G. R. Hudson, "Program Errors as a Birth-and-Death Process," SDC Report SP-3011, Dec. 4, 1967.

25. J. S. Coutinho, "Software Reliability Growth," 1973 IEEE Symposium on Computer Software Reliability, New York City, pp. 58-64.

26. W. Feller, "An Introduction to Probability Theory and Its Application," Vol. I, Second Edition, John Wiley and Sons, Inc., New York, 1957, pp. 43-44.

27. M. Lipow, "Estimation of Software Package Residual Errors," TRW-SS-72-09, Nov. 1972.

28. B. Rudner, "Seeding/Tagging Estimations of Software Errors: Models and Estimates," RADC-TR-77-15, January 1977, A036655.

29. E. C. Nelson, "Software Reliability," TRW-SS-75-05, Nov. 1975.

30. E. C. Nelson, "Software Reliability," 1975 International Symposium on Fault-Tolerant Computing FTC-5, Paris, France, June 18-20, 1975, pp. 24-28.

31. E. C. Nelson, "Software Verification and Validation," Proceedings of TRW Symposium on Reliable, Cost-Effective, Secure Software, TRW-SS-74-14, March 1975, pp. 5-18, 38.

32. T. H. Thayer, "Software Reliability Study," RADC-TR-76-238, Final Technical Report, August 1976, A030798.

33. J. R. Brown and M. Lipow, "Testing for Software Reliability," Proc. 1975 International Conference on Reliable Software, pp. 518-527.

34. W. H. MacWilliams, "Reliability of Large Real-Time Control Software Systems," pp. 1-6, Proc. 1973 IEEE Symposium on Computer Software Reliability, New York City, 1973.

35. R. C. Cheung, P. C. Tam, and S. S. Yau, "A User-Oriented Software Reliability Model Using Self-Metric Software, " RADC Technical Report August, 1977.